Introducción a la algorítmica

Jordi Àlvarez Canal
Josep Vilaplana Pastó

PID_00149893

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Este módulo introduce el lenguaje algorítmico que utilizaremos para expresar nuestros primeros algoritmos. En los módulos siguientes veremos, a medida que vayamos introduciendo otros conceptos, ampliaciones de este lenguaje.

El aprendizaje de la programación empieza por llevar a cabo acciones sencillas con algunas herramientas básicas y, una vez asimiladas, progresamos acumulando nuevos conocimientos más complejos y avanzados con las herramientas correspondientes. Es necesario hacer este proyecto de forma gradual y a medida que los conceptos más sencillos están bien asimilados. Por eso, es importante dominar los temas de cada módulo y, especialmente de éste, que es la base de los que vienen a continuación.

Algunos de vosotros todavía os preguntaréis por qué tenemos que utilizar un lenguaje diferente del que usamos habitualmente: catalán, castellano, inglés, etc. Así, por ejemplo, tenemos que un libro de recetas de cocina es en realidad un libro de algoritmos, donde el estado inicial de cada algoritmo presentado consiste en tener los ingredientes al alcance (y los recursos necesarios para cocinarlos) y el estado final es la presencia en la mesa de un plato con el que nos podemos chupar los dedos (o eso dice el autor del libro).

Un libro de cocina está escrito en lenguaje natural: catalán, castellano, inglés... Así pues, ¿por qué no describimos también nosotros los algoritmos en lenguaje natural?

Se dan dos importantes factores para no hacerlo. En primer lugar, el lenguaje natural presenta a menudo ambigüedades que dificultan la comprensión del mensaje que debemos transmitir. Esto nos podría llevar muchas veces a malas interpretaciones de algoritmos. Evidentemente, podemos intentar no ser ambiguos y lo podemos conseguir (con bastante esfuerzo algunas veces).

Sin embargo, hay un factor más importante. Imaginad que queremos hacer una *fideuà* (guiso de fideos con caldo de pescado). Como precondición, tendremos que disponer de todos los ingredientes, utensilios necesarios y elementos combustibles (gas, leña, etc.). Hasta aquí, ningún problema.

Comenzamos a describir el algoritmo. En primer lugar tenemos que trocear media cabeza de ajos. ¿Qué significa trocear? ¿Qué medida deben tener los trocitos resultantes? Más adelante, el algoritmo nos indica que tenemos que poner una sartén con cuatro cucharaditas de aceite de oliva en el fuego, entre bajo y medio, y freír el ajo que tenemos troceado. ¿Qué quiere decir exactamente entre bajo y medio? Cuando éste esté un poco dorado (no demasiado), tenemos que añadirle el tomate (que tendremos pelado y troceado previamente). Bien, ¿cuándo está el ajo un poco dorado, pero no mucho?

Todas estas preguntas las puede responder con seguridad una persona que esté acostumbrada a cocinar; pero una persona que no haya cocinado nunca en su vida será incapaz de ejecutar el algoritmo correctamente (será incapaz incluso de entenderlo).

Entonces, ¿hasta qué nivel tenemos que describir el algoritmo? Evidentemente, depende de a quién se lo estemos explicando. Éste es el segundo problema importante con que nos encontraremos si utilizamos el lenguaje natural para describir algoritmos.

Por estas dos razones (ambigüedad y dependencia del interlocutor), necesitamos definir otro lenguaje que disponga de una sintaxis reducida, simple y precisa que permita expresar algoritmos sin ambigüedades y con la claridad y precisión necesarias para que puedan ser interpretados por cualquier procesador sin presuponer ningún conocimiento previo por su parte. Éste será nuestro **lenguaje algorítmico.**

También debéis daros cuenta de que utilizar el lenguaje algorítmico para describir nuestros diseños hace que, al mismo tiempo, obtengamos una ventaja adicional. Este lenguaje teórico nos da la suficiente generalidad como para que lo podamos utilizar para describir cualquier tipo de algoritmo sin tener que preocuparnos de su posterior codificación en un entorno concreto. Si aprendemos a diseñar algoritmos en este lenguaje genérico, después no nos debe costar demasiado codificarlos en el lenguaje de programación que el entorno en que estamos trabajando requiera, siempre que se trate de un lenguaje imperativo y procedimental.

Pero recordad que...

... existen otros tipos de programación, como por ejemplo la programación declarativa (donde encontraremos lenguajes como Lisp y Prolog). En esta asignatura estudiaremos la programación imperativa.

En el mercado hay muchos lenguajes de programación y, seguramente, necesitaréis utilizar más de uno (incluso algunos que todavía ni siquiera existen) en vuestra práctica profesional. Una asignatura de programación como ésta pretende que aprendáis a programar en el paradigma de la programación imperativa, independientemente del lenguaje en que finalmente lo hagamos. Ésta es precisamente la otra ventaja que nos da el lenguaje algorítmico: es suficientemente genérico como para que la metodología que aprenderemos no esté relacionada con ningún lenguaje concreto y, a la vez, para que nos permita fácilmente codificar los diseños en cualquiera de los lenguajes de programación estructurada que se encuentran en el mercado. Los diseños que hacemos nos servirán siempre para aplicarlos al lenguaje de programación que tengamos que utilizar en cada momento, sólo tendremos que aprender las particularidades del lenguaje de programación concreto.

Finalmente, este módulo también nos introduce en la especificación de algoritmos. Recordad que ya hemos dicho en el módulo de introducción que la especificación es un paso previo que nos ayudará a entender qué problema tenemos que resolver antes de ponernos a hacer el diseño del algoritmo. Como veremos, la especificación de los algoritmos será relevante para asegurarnos de que nuestros algoritmos resuelven exactamente los problemas para los que han sido planteados.

Objetivos

Una vez hayamos estudiado este módulo, tendremos que ser capaces de:

1. Conocer los elementos básicos del lenguaje algorítmico, que nos debe servir para desarrollar algoritmos.

2. Evaluar mentalmente una expresión escrita en lenguaje algorítmico, teniendo en cuenta las precedencias de los diferentes operadores. Ser capaces, de esta manera, de “redactar” una expresión en lenguaje algorítmico, sabiendo cuándo tenemos que utilizar paréntesis, según la precedencia de los operadores.

3. Entender qué quiere decir especificar un problema, conociendo el significado de los términos precondición y postcondición. Saber distinguir entre especificación y algoritmo.

4. Ante un problema, ser capaces de especificarlo.

5. Conocer las diferentes construcciones que nos ofrece el lenguaje algorítmico para desarrollar algoritmos. Dado un algoritmo sencillo, como los que aparecen en este módulo, ser capaces de averiguar qué hace.

6. Definir nuevas acciones y funciones y, una vez definidas, saberlas invocar. Saber distinguir entre parámetro real (o actual) y parámetro formal, así como entre parámetro de entrada, entrada/salida, y salida.

7. Desarrollar algoritmos sencillos usando las herramientas que nos ofrece el lenguaje algorítmico.

1. Objetos elementales del lenguaje algorítmico

Aunque todavía no sabemos cómo diseñar algoritmos, sí sabemos que tendremos que tratar con datos. Al diseñar un algoritmo, necesitaremos referenciar estos datos de alguna forma y explicitar qué comportamientos esperamos de los mismos. Para conseguir este objetivo, introducimos el concepto de objeto, que será el soporte para mantener los datos que trate un algoritmo.

A menudo, cuando pensamos en un objeto concreto, nos viene a la cabeza, entre otras cosas, un abanico de acciones u operaciones posibles que nos pueden llevar hasta el objeto, y otras acciones que no podemos hacer con éste.

Un objeto tiene tres atributos:

- Nombre o identificador
- Tipo
- Valor

El nombre nos permite identificar unívocamente el objeto. El tipo indica el conjunto de valores que puede tener y qué operaciones se pueden aplicar sobre el objeto.

Por ejemplo,...

... imaginemos que tenemos el objeto semáforo de un tipo que sólo acepta los valores verde, rojo, amarillo y apagado. El valor del objeto semáforo en algún momento dado será rojo. En otro instante será verde, etc. En cambio, el objeto señal de tráfico, como el de la dirección prohibida, tendrá siempre un valor constante dentro del tipo de valores indicadores de señales de tráfico.

El valor de un objeto no tiene que ser necesariamente un número. El valor de un objeto será un elemento del conjunto al que pertenece y que viene indicado por su tipo correspondiente. Además, encontraremos el caso de objetos cuyo valor podrá ser cambiado por otro del mismo tipo.

Por tanto, un objeto podrá ser:

- **Constante**: su valor no es modificable.
- **Variable**: su valor se puede modificar.

1.1. Tipos elementales

Pasemos a ver qué tipos presenta inicialmente el lenguaje para poder declarar nuestros objetos. Son los llamados tipos elementales, y son cuatro: **booleano**, **carácter**, **entero** y **real**. Más adelante veremos que el lenguaje nos permitirá definir nuestros tipos particulares.

En el módulo "Estructuras de datos" veremos la posibilidad de construir otros objetos más complejos.

Para describir el tipo, expondremos el identificador o nombre en que el lenguaje lo reconoce, cuáles son los valores que puede tomar, qué operaciones puede realizar y cuál es la sintaxis reconocida por el lenguaje de sus valores.

1.1.1. Tipo booleano

Es uno de los tipos más utilizados; su origen se encuentra en el álgebra de Boole. En muchos casos, sus dos valores sirven para indicar la certeza o falsedad de un estado concreto del entorno. También se conoce como *tipo lógico*.

El lenguaje algorítmico lo reconoce mediante el identificador *booleano*. Sólo tiene dos valores posibles: **cierto** y **falso**. El lenguaje algorítmico reconocerá las palabras **cierto** y **falso** como los valores booleanos. Las operaciones internas (aquellas que devuelven un valor del mismo tipo) son **no** (negación), **y** (conjunción), **o** (disyunción).

Los operadores lógicos **no**, **y**, **o** (negación, "y" lógica, "o" lógica respectivamente) se definen mediante la siguiente tabla:

a	*b*	no *a*	*a* y *b*	*a* o *b*
cierto	cierto	falso	cierto	cierto
cierto	falso	falso	falso	cierto
falso	cierto	cierto	falso	cierto
falso	falso	cierto	falso	falso

En la siguiente tabla tenemos las características del tipo booleano:

Identificador	Booleano
Rango de valores	cierto, falso
Operadores internos	no, y, o, =, ≠, <, ≤, >, ≥
Operadores externos	
Sintaxis de valores	cierto, falso

Prestad atención al hecho de que en el apartado de operadores internos hemos introducido otros símbolos, los operadores relacionales (=, ≠, <, ≤, >, ≥), que os explicaremos a continuación en el siguiente apartado.

Operadores relacionales

Introducimos otros operadores con los que posiblemente no estéis familiarizados. Son los operadores relacionales y, como veremos más adelante, pueden operar sobre otros tipos.

Los operadores =, ≠, <, ≤, >, ≥ se denominan operadores relacionales y se aplican a dos operandos **del mismo tipo**. Además, los operadores relacionales <, ≤, >, ≥ sólo se pueden aplicar si el tipo tiene una relación de orden. Por convenio, en el tipo booleano se tiene que:
falso < **cierto**

Así pues, **falso** < **cierto** es una operación que da como valor **cierto**. La operación **cierto** = **falso** da el valor de **falso**, mientras que **cierto** ≤ **cierto** da el valor de **cierto**, etc.

A continuación, tenéis una tabla para los operadores =, <, >:

A	*b*	*a* = *b*	*a* < *b*	*a* > *b*
cierto	**cierto**	**cierto**	**falso**	**falso**
cierto	**falso**	**falso**	**falso**	**cierto**
falso	**cierto**	**falso**	**cierto**	**falso**
falso	**falso**	**cierto**	**falso**	**falso**

Las siguientes expresiones cubrirán el resto de casos que podemos encontrar con los operadores relacionales:

- $a \neq b$ es equivalente a **no** $(a = b)$.
- $a \geq b$ es equivalente a $(a > b)$ **o** $(a = b)$.
- $a \leq b$ es equivalente a $(a < b)$ **o** $(a = b)$.

Los operadores...

... relacionales no están restringidos al tipo booleano, sino que, como veremos más adelante, se pueden aplicar a otros tipos. El resultado de aplicar un operador relacional sobre otros tipos nos dará un resultado booleano

1.1.2. Tipo carácter

Nosotros nos podemos comunicar mediante textos que se construyen a partir de unos símbolos llamados caracteres. El tipo más elemental para que un algoritmo pueda gestionar información simbólica es el tipo carácter.

El identificador del tipo es *caracter*. La sintaxis que reconoce el lenguaje algorítmico de sus valores es el propio carácter delimitado por comillas simples. Por ejemplo, 'e' es el valor de carácter correspondiente a la letra "e", '=' es el valor correspondiente al símbolo de igual, y ' ' es el carácter correspondiente al espacio.

El rango de valores que puede tener este tipo depende de la codificación utilizada por el computador. En cualquier caso, será un conjunto finito de valores.

Hay muchos códigos...

... de caracteres en el mundo de las computadoras.

Algunos códigos son específicos de un fabricante concreto, y otros vienen establecidos por organismos oficiales nacionales y/o internacionales. La codificación ASCII (que satisface el mundo inglés) tiene 128 valores y es una de las más utilizadas en todo el mundo. En Europa está teniendo una amplia difusión el código iso-latin de 256 caracteres. Otras codificaciones, como la Unicode, reúnen la mayoría de los alfabetos existentes. En principio, el uso de un código u otro no nos debe preocupar, teniendo en cuenta que nosotros denotaremos los caracteres por su representación gráfica.

Dentro de este conjunto de caracteres encontramos definida una relación de orden. Esto nos permite aplicar los operadores relacionales externos al conjunto. En el caso de los caracteres que representan las letras que conocemos, el orden sigue un criterio alfabético. Por ejemplo, el valor de la operación 'a' < 'b' es **cierto**, 'z' < 'a' es **falso**. Notad que el carácter 'A' es diferente de 'a' y, por tanto, 'A' = 'a' es **falso.** Podemos hablar, por lo tanto, de un orden alfabético entre los caracteres que representan las letras minúsculas, y de otro orden entre los caracteres que representan las mayúsculas. También podremos hablar de un orden entre los caracteres dígitos ('0', '1', ... '9'). Por tanto, '0' < '5' es **cierto**.

Prestad atención al hecho de que hablamos de órdenes entre subconjuntos de caracteres que nos pueden ser útiles. También se establece un orden entre estos conjuntos que corresponde a un convenio de la codificación utilizada para

representar los caracteres (observad el margen). En el caso de utilizar la codificación ASCII, las minúsculas siempre son mayores que las mayúsculas, y las mayúsculas, mayores que los caracteres dígitos.

Con los caracteres...

... acentuados no hay un consenso entre los códigos sobre su orden y, por tanto, no podemos decir nada común al respecto. Notad que 'a' = 'á' es **falso**, etc.

En resumen,

Identificador	**Caracter**
Rango de valores	Conjunto finito de valores que dependen de un código usado por el computador.
Operadores internos	
Operadores externos	=, ≠, <, ≤, >, ≥
Sintaxis de los valores	Ejemplos: '*a*' '*W*' '1'. Observad que el carácter '1' no tiene nada que ver con el entero 1.

1.1.3. Tipo entero

Hubiera estado bien hablar del tipo número tal como nosotros lo entendemos. En programación tendremos que distinguir los enteros de los reales y, aunque en la vida real consideremos los enteros como un subconjunto de los reales, en programación los trataremos como dos conjuntos diferentes que no guardan ninguna relación. Esto se debe a la representación interna de cada tipo.

El tipo entero se identifica con el nombre reservado **entero**. A pesar de esto, dado que los ordenadores tienen memoria finita, no podremos tener todos los que queramos, sino un subconjunto finito que estará limitado por un entero mínimo *ENTEROMIN* y un entero máximo *ENTEROMAX*. Los valores *ENTEROMIN* y *ENTEROMAX* dependerán del computador que utilicemos. La sintaxis de los valores vendrá dada por el conjunto de cifras que representa el valor entero precedido del signo menos (–) cuando el entero es negativo. Así, 4 es un entero, 4.876 es otro entero, y –10 es otro entero. Notad que '5' no es un entero, sino el carácter '5'. Las operaciones internas que se pueden hacer con el entero son: el cambio de signo (–), la suma (+), la resta (–), la multiplicación (∗), la división entera (**div**), y el resto de la división o módulo (**mod**).

Las operaciones **div** y **mod** son las que os pueden resultar más extrañas. A continuación tenéis ejemplos de la operación **div**:

8 **div** 4 = 2, 7 **div** 2 = 3 y 3 **div** 8 = 0

Ejemplos de la operación **mod**:

8 **mod** 4 = 0, 7 **mod** 2 = 1 y 3 **mod** 8 = 3



Evidentemente, tendremos que 0 ≤ dividendo **mod** divisor < divisor.

Las operaciones **div** y **mod** están definidas por el teorema de la división entera de Euclides:

Para todo dividendo positivo y divisor positivo y diferente de 0, la ecuación:

dividendo = divisor ∗ cociente + resto

tiene un único valor de cociente y un único valor de resto, si el resto cumple que es mayor o igual que 0, y a la vez es menor que el divisor (0 ≤ resto < divisor)

Y estos valores únicos son cociente = dividendo **div** divisor, y

resto = dividendo **mod** divisor

La división entera por 0 producirá un error.

Evidentemente, se establece una relación de orden entre los enteros que permite utilizar los operadores relacionales (4 < 10 es **cierto**, 235 > 4.000 es **falso**)

En resumen:

Identificador	**Entero**
Rango de valores	*Desde ENTEROMIN* hasta *ENTEROMAX*.
Operadores internos	− (Cambio signo), +, −, ∗, **div, mod**
Operadores externos	Relacionales: =, ≠, <, ≤, >, ≥
Sintaxis de valores	Ejemplos: 3, 465.454, −899.993

1.1.4. Tipo real

Como hemos adelantado en el último apartado, otro tipo para representar números es el tipo real. Al igual que los enteros, estará limitado por un valor mínimo (*REALMIN*) y un valor máximo (*REALMAX*). Estos valores dependerán de la representación interna del computador. Además, sólo se puede representar un número finito de valores del rango. Pensad que entre dos números reales cualesquiera, hay un número infinito de reales, y el ordenador es finito.

Puesto que sólo se puede representar un conjunto finito de reales, los programas que trabajan con reales están sometidos a problemas de precisión y errores en los cálculos. Desde un punto de vista informático, los enteros no tienen nada que ver con los reales, ya que el comportamiento en el computador de ambos tipos, así como sus respectivas representaciones internas en el computador, son diferentes.

Los reales comparten con los enteros los símbolos de operadores +, −, ∗ (suma, resta, producto). A pesar de la coincidencia de símbolos, es necesario que consideréis los símbolos como operaciones diferentes de las que se aplican a los enteros, en el sentido que por ejemplo, la suma de reales dará un resultado real, mientras que la suma de enteros dará un resultado entero. También tienen la operación de división (/). Notad que la división por 0 produce un error.

En las asignaturas...

... relacionadas con estructuras y arquitecturas de computadoras, podréis profundizar más en la problemática de los reales.

La sintaxis de valores se lleva a cabo mediante una sucesión de dígitos, donde aparece el carácter "." para denotar la coma decimal. Por ejemplo,

49.22, 0.5, 0.0001

Para representar *10 elevado a* algo, utilizaremos el carácter *E*. Por ejemplo, 5.0E-8 corresponde a $5 \cdot 10^{-8}$ o 0.00000005.

Observad que la sintaxis de reales se diferencia de la de enteros por la presencia del punto. Así, si vemos escrito *5*, sabremos que corresponderá al entero 5, mientras que si vemos escrito *5.0*, sabremos que corresponde al real 5. A pesar de que para nosotros corresponde al mismo número, el computador lo tratará de forma diferente.

En resumen:

Identificador	Real
Rango de valores	Entre *REALMIN* y *REALMAX*, y no todos los valores que están en el rango.
Operadores internos	– (cambio de signo) +, –, *, /
Operadores externos	Relacionales: =, ≠, <, ≤, >, ≥
Sintaxis de valores	Ejemplos: 0.6, 5.0E–8, 49.22E + 8, 1.0E5, 4.0

1.2. Declaración de objetos

Antes de que un algoritmo utilice un objeto, es preciso que éste se haya declarado previamente.

Si el objeto es constante, se debe declarar dentro de un apartado acotado por las palabras clave **const**...**fconst**. Si es un objeto variable, se declarará dentro de un apartado acotado por las palabras **var**...**fvar**.

En el caso de un objeto constante, tenemos que escribir su nombre, su tipo y el valor constante:

```
const
    nombre : tipo = valor
fconst
```

En el caso de que el objeto sea variable, la declaración toma la forma:

```
var
    nombre : tipo
fvar
```

En el identificador *nombre* es necesario que el primer carácter sea siempre un carácter alfabético. El resto de caracteres que siguen pueden ser alfabéticos, numéricos o el carácter "_". No puede haber espacios entre caracteres.

Por ejemplo, el nombre 1*xl* es incorrecto, el nombre *Jose-Maria* también es incorrecto; mientras que *xl*1 es correcto, *Jose_Maria* es correcto.

En el caso de variables del mismo tipo, se pueden declarar de la siguiente forma:

```
var
    nombre1, nombre2, nombre3, ...: tipo;
fvar
```

Como podéis comprobar, existe una serie de palabras que delimitan las partes del algoritmo. Estas palabras se denominan **palabras clave**, y nos ayudan a identificar fácilmente a qué se corresponde cada elemento. Estas palabras clave son palabras exclusivas del lenguaje algorítmico, por lo que no las podemos usar como nombres de los objetos (constantes o variables). Por ejemplo, no podemos poner *var* como nombre de una constante o variable.

Las palabras clave aparecerán en negrita en nuestros algoritmos para diferenciarlas del resto.

Se admite una excepción a esta regla. Aunque hemos definido el operador lógico **y** y, puesto que como operador del lenguaje es una palabra reservada, se acepta la posibilidad de definir una variable con el nombre *y*.

Para distinguirlas, el operador *y* aparecerá en negrita como palabra reservada que es.

Ejemplos

```
const
   PI: real = 3.1415926535897932;
   RAIZ_2: real = 1.4142135623730950;
   LN_2: real = 0.6931471805599453;
   PtsEURO: real = 166.386;
   DiasSemana: entero = 7;
   separador: caracter = '–'
fconst

var
   Saldo, Precio, NumeroArticulos: entero;
   DiametroCirculo: real;
   CaracterLeido: caracter
fvar
```

1.3. Expresiones

Cuando escribamos algoritmos, tendremos la necesidad de operar entre los diferentes objetos que hayamos declarado por nuestro entorno de trabajo para obtener algún valor concreto. Haremos esto mediante las expresiones.

Una **expresión** es cualquier combinación de operadores y operandos.

Un operando puede ser una variable, una constante o una expresión. Los operadores pueden ser unarios o binarios. Un operador unario es aquel que sólo opera con un único operando. Es, por ejemplo, el caso de la negación lógica y el signo menos aplicado a un número. Un operador binario es aquel que opera con dos operandos, por ejemplo, la suma o la resta.

La definición anterior de expresión no es suficiente. Para que una expresión nos sea útil, es necesario que ésta siga unas reglas sintácticas y que se respete el significado de los símbolos utilizados. En otras palabras, hace falta que una expresión sea correcta en sintaxis y semántica.

1.3.1. Sintaxis

Una expresión correcta y, por tanto, evaluable debe seguir las siguientes normas de combinación:

- Un valor es una expresión.
- Una variable es una expresión.
- Una constante es una expresión.
- Si E es una expresión, (E) también lo es.
- Si E es una expresión y $\cdot$ es un operador unario, $\cdot\, E$ también lo es.
- Si E_1 y E_2 son expresiones y $\cdot$ es un operador binario, $E_1 \cdot E_2$ también lo es.
- Si $E_1, E_2, ..., E_n$ son expresiones y f es una función, $f(E_1, E_2, ..., E_n)$ también lo es.

Las expresiones se escriben linealizadas, es decir, de izquierda a derecha y de arriba abajo.

Por ejemplo, para a,b,c,d,e enteros,

$$\frac{\frac{a}{b}}{\frac{c}{d}} \times e = (a \text{ div } b) \text{ div } (c \text{ div } d) \times e$$

Notad que el signo...

... menos (–) es un símbolo que tiene dos significados: cuando aparece delante de un solo número, significa cambio de signo (operador unario). Cuando está entre dos números, se refiere a la resta de los dos números (operador binario).

Por el momento,...

... pensad en funciones matemáticas. Más adelante definiremos el concepto de función dentro de la algorítmica.

Semántica

Aunque una expresión tenga una sintaxis correcta, ésta puede carecer de sentido, ya que hay que respetar la tipificación de variables, constantes, operadores y funciones. Hablaremos, pues, de corrección sintáctica y de corrección semántica.

Por ejemplo,...

... la expresión *a* **div** *b* es correcta sintácticamente. Si *a* y *b* son variables enteras, la expresión será correcta semánticamente, teniendo en cuenta que la operación **div** está definida para los enteros. Si *a* y *b* son booleanos (o reales, o caracteres), la expresión será incorrecta semánticamente, ya que no tiene sentido.

Por ejemplo, si encontramos un operador suma aplicado a variables booleanas, la expresión puede ser correcta sintácticamente, pero será incorrecta semánticamente, ya que la suma no está definida para los booleanos. Otro ejemplo es 4.0 + 7, la cual no es correcta semánticamente porque mezcla dos tipos de

datos diferentes. Notad que la última expresión puede tener sentido fuera del contexto algorítmico, pero no lo tiene dentro de la algorítmica.

1.3.2. Evaluación

La evaluación de una expresión se hace de izquierda a derecha comenzando por las funciones (que ya veremos más adelante) y los elementos comprendidos entre los paréntesis más internos. Dentro de los paréntesis se evaluarán primero las operaciones más prioritarias y después, las de menos prioridad.

Una constante se evaluará según el valor descrito en su definición, así como según el valor que guarda en el momento en que se evalúa.

Las prioridades de los operadores, ordenados de más a menos prioritario, son:

1. – (cambio de signo), **no**
2. ∗, /, **div**, **mod**
3. +, – (resta)
4. =, ≠, <, ≤, >, ≥
5. **y**
6. **o**

En igualdad de prioridades, se evalúa de izquierda a derecha.

Notad, sin embargo,...

... que con un uso continuo de operadores unarios, se evalúan de derecha a izquierda (al revés de como se propone). Por ejemplo, – – – –5 = –(–(–(–5))) = 5

Ejemplos de evaluación de expresiones:

1. *c* **div** *d* ∗ *e*. Primero se evalúa *c* **div** *d*, y el resultado de esta expresión se multiplica por *e*. La expresión es equivalente a (c **div** *d*) ∗ *e*, y NO tiene nada que ver con *c* **div** (*d* ∗ *e*).

2. 3 ∗ 1 **div** 3 = 1 porque es equivalente a (3 ∗ 1) **div** 3 = 1

3. 3 ∗ (1 **div** 3) = 0

4. (*a* ∗ *b*)–((*c* + *d*) **div** *a*) **mod** (2 ∗ *a*)

El orden de evaluación es el siguiente:

1. *a* ∗ *b*
2. *c* + *d*
3. (*c* + *d*) **div** *a*
4. 2 ∗ *a*
5. El valor del paso 3 **mod** el valor del paso 4
6. El valor del paso 1 menos el valor del paso 5

5. *a* + *b* = *a* * *d* **div** *c*

El orden de evaluación de esta expresión es:

Se evalúa *a* * *d*
Se evalúa el resultado del paso 1 **div** *c*

Se evalúa *a* + *b*
Se evalúa = con el resultado del paso 2 y el resultado del paso 3. El resultado de evaluar esta expresión es un booleano.

6. $a > b = c$

El orden de evaluación de esta expresión es:

Se evalúa *a* > *b*
Se evalúa = con el resultado del paso 1 y *c*. El resultado de evaluar esta expresión es un booleano.

7. *a* * *b* > *c* + *d* **o** *e* < *f*

El orden de evaluación de esta expresión es:

Se evalúa *a* * *b*
Se evalúa *c* + *d*
Se evalúa > con el resultado del paso 1 y el resultado del paso 2
Se evalúa el resultado *e* < *f*
Se evalúa **o** con el resultado del paso 3 y el resultado del paso 4. El resultado de evaluar esta expresión es un booleano.

8. **no** *a* = *b* **y** $c \geq d$

El orden de evaluación de esta expresión es:

Se evalúa **no** *a*
Se evalúa = con el resultado del paso 1 y *b*
Se evalúa el resultado $c \geq d$
Se evalúa **y** con el resultado del paso 2 y el resultado del paso 3. El resultado de evaluar esta expresión es un booleano.

1.4. Definición de tipos. Tipos enumerativos

La notación algorítmica nos permite definir nuevos tipos para poder trabajar con objetos que se encuentren más cercanos al problema que estamos tratando.

Un constructor de tipos es una construcción de la notación algorítmica que permite definir nuevos tipos.

En el módulo "Estructuras de datos" veremos cómo construir nuevos tipos a partir del constructor de tipos.

En este apartado veremos el **constructor de tipos por enumeración**, que permite definir un nuevo tipo de forma que los valores posibles que pueda tener están enumerados en la propia definición del tipo.

La sintaxis es:

tipo
nombre = {$valor_1$, $valor_2$, ..., $valor_n$}
ftipo

Los únicos operadores que se pueden aplicar a los tipos enumerativos son los operadores relacionales externos. El orden en que se enumeran los valores es el que se utiliza para establecer las relaciones de menor y mayor.

tipo
color = {verde, azul, rojo, amarillo, magenta, cyan, negro, blanco};
día = {lunes, martes, miércoles, jueves, viernes, sábado, domingo};
mes = {enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre, noviembre, diciembre};
ftipo

La expresión *febrero* < *marzo* nos devolverá el valor **cierto.**

1.5. Funciones de conversión de tipos

A veces, necesitaremos pasar de un tipo de valor a otro tipo. Para hacerlo, dispondremos de unas funciones predefinidas que nos permitirán convertir un valor de un cierto tipo elemental en otro. Las funciones son:

- *realAEntero*: esta función acepta un argumento de tipo real y lo convierte a tipo entero. Es necesario tener en cuenta que el nuevo valor resultante de la conversión pierde todos los decimales que pueda tener el valor real. Por ejemplo, *realAEntero*(*4.5768*) nos devuelve el valor de tipo entero 4.

- *enteroAReal*: esta función acepta un argumento de tipo entero y lo convierte a tipo real. Por ejemplo, *enteroAReal*(*6*) nos devuelve el valor de tipo real 6.0. Recordad que, desde el punto de vista algorítmico, 6.0 no tiene nada que ver con 6.

- *caracterACodigo*: esta función acepta un argumento de tipo carácter y lo convierte a tipo entero. El entero tiene que ver con la codificación que el carácter tiene internamente en el computador. La función se utiliza en casos muy especiales y poco frecuentes. Por ejemplo, *caracterACodigo*('A') nos devolverá el valor 65 si utilizamos la codificación ASCII, o 193 si utilizamos la codificación EBCDIC. Con otros códigos tendrá otros valores.

- *codigoACaracter*: esta función acepta un argumento de tipo entero y lo convierte a tipo carácter. A pesar de esto, el efecto de la función está limitado a un subconjunto del tipo entero que tiene que ver con la codificación de caracteres que utiliza el computador. Así pues, la función no tiene un efecto definido para aquellos enteros que no correspondan a un código de carácter. Por ejemplo, en la codificación ASCII utilizada en los ordenadores personales, el subconjunto de enteros es el intervalo entre 0 y 127. Por ejemplo, *codigoACaracter(10)* nos retornará un carácter no visualizable en la codificación ASCII que corresponde al salto de línea en un texto. En cambio, por la misma codificación, c*odigoACaracter(48)* nos devolverá el carácter “0”.

Puesto que...

... la codificación ASCII es sólo para el caso del idioma inglés, para otros idiomas existe una extensión del ASCII que va de 128 a 255. No obstante, esta extensión no está del todo difundida, y además cambia de acuerdo con el idioma que se quiera utilizar (francés, español, etc.).



Veremos la sintaxis de estas funciones en el último apartado de este módulo.

2. Especificación de algoritmos

2.1. Algoritmo y cambio de estado

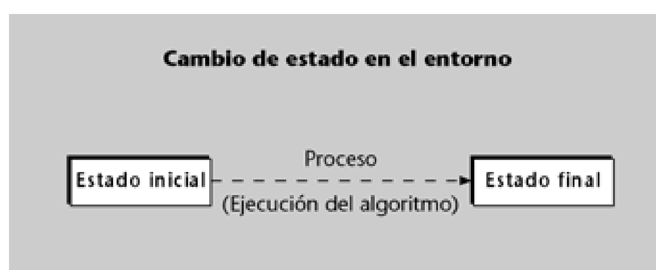
Un algoritmo es una descripción de un método sistemático para resolver un problema. La forma como un algoritmo resuelve un problema forma parte del algoritmo mismo; pero en cualquier caso, la resolución del problema se consigue actuando sobre el entorno donde está inmerso el algoritmo, modificándolo convenientemente.

Así, por ejemplo, una lavadora dispone de un método sistemático para lavar la ropa. Por tanto, podemos concluir que las lavadoras disponen de un algoritmo para lavar la ropa, tal como hemos visto en el módulo anterior.

- **¿Cuál es, en este caso, el entorno sobre el que actúa la lavadora?** Pues, de una manera simplificada, podemos decir que es la ropa que se encuentra dentro del tambor de la lavadora.

- **¿Cómo actúa?** El proceso que utiliza la lavadora lo tenéis descrito en el módulo anterior. Pero lo que realmente nos importa es que inicialmente la ropa está (más o menos) sucia; y después de que la lavadora aplique el algoritmo de lavar, la ropa está limpia.

Es decir, se ha producido un **cambio de estado** en el **entorno** sobre el que actúa la lavadora. El **estado inicial** era que la ropa estaba sucia, y el **estado final** es que la ropa está limpia. Por tanto, hemos solucionado el problema, que era lavar la ropa.



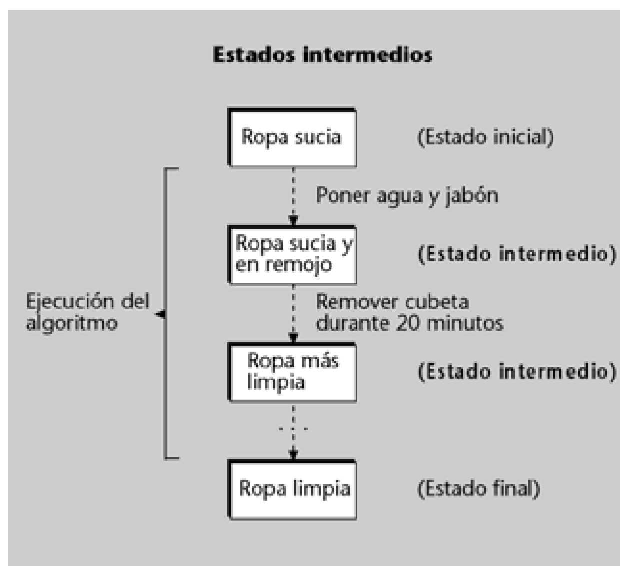
Este cambio de estado entre el estado inicial y el estado final no tiene por qué producirse de repente. Hemos visto en el módulo anterior que en el algoritmo que utiliza la lavadora para lavar ropa es necesaria la ejecución de un conjunto de acciones. Cada una de estas acciones modifica el estado de alguna manera determinada. Es decir, entre el estado inicial y el estado final pueden darse muchos **estados intermedios**.

Recordad el algoritmo para lavar la ropa del módulo "Introducción a la programación".

Así, en primer lugar, la lavadora llena su tambor de agua mezclada con jabón. Por tanto, la ropa pasa de estar sucia a estar sucia y en remojo con agua y jabón. Posteriormente, el tambor se mueve durante 20 minutos. Esto hace que

la suciedad de la ropa, al estar en remojo con agua y jabón pasa, en gran parte, a estar en el agua. Después de estos 20 minutos, la ropa está en remojo con agua y jabón, la ropa está más limpia y el agua está más sucia.

Podríamos seguir analizando el resto de acciones que ejecuta la lavadora para lavar la ropa. Todas producirían cambios de estado sobre el entorno que, basándose en el estado inmediatamente anterior, nos irían acercando hacia el estado final deseado.



Traducimos ahora todo esto al mundo del lenguaje algorítmico, que es donde trabajaremos en esta asignatura.

Como ya se dijo en el módulo anterior, el **entorno** de un algoritmo es el conjunto de objetos que intervienen en éste. En el apartado anterior habéis visto con qué objetos trabaja el lenguaje algorítmico.

De todos éstos, los únicos objetos que la ejecución de un algoritmo podrá modificar son las **variables**. Así pues, un proceso (la ejecución de un algoritmo) sólo puede modificar el entorno cambiando el valor de las variables que éste contiene.

Por este motivo, decimos que el **estado** en un momento determinado de la ejecución de un algoritmo nos viene determinado por el valor de las variables que intervienen.

2.2. ¿Qué significa *especificar*?

El primer paso en la construcción de un algoritmo consiste en saber claramente qué es lo que queremos resolver, cuál es el problema que tenemos entre manos. De esta forma, podremos intentar diseñar un algoritmo que lo resuelva. Esto lo haremos mediante la especificación.

Especificar consiste en dar la información necesaria y suficiente para definir el problema que hay que resolver de la manera más clara, concisa y no ambigua posible.

Tenéis que distinguir claramente entre *especificación* y *algoritmo*:

- La **especificación** nos dice cuál es el problema que tenemos que resolver.
- El **algoritmo** nos dice cómo resolvemos el problema.

Tal como hemos visto en el apartado anterior, dado un problema que queremos resolver y un algoritmo que lo resuelve; lo que hace éste es modificar el entorno convenientemente, de forma que obtengamos el resultado deseado.

Así pues, especificaremos dando una descripción lo más clara, concisa y no ambigua posible de los siguientes aspectos:

- El estado inicial en que se encuentra el entorno y del cual partimos.
- El estado final del entorno al que debemos llegar para resolver el problema.

La descripción del estado inicial se denomina **precondición**, y la descripción del estado final, **postcondición**.

Fijaos en que vemos el algoritmo como una caja negra que, a partir de un estado inicial determinado por la precondición, obtiene el estado final descrito en la postcondición. Cuando especificamos un problema, sólo nos importan los estados inicial y final, y no todo el conjunto de estados intermedios por los que se pasa hasta llegar al estado final. Es más, dada una especificación de un problema, puede haber más de un algoritmo que solucione el problema (de la misma forma que para ir de un lugar a otro podemos ir por varios caminos).

Notad que...

... cuando especificamos un problema, estamos reestructurando la información contenida en el enunciado de este problema. Con la especificación podemos apreciar con más claridad cómo la resolución del problema consiste en el paso de un estado a otro.

Tenéis que tener bien claro qué forma parte de lo *que tenemos que resolver* y qué forma parte del *cómo lo resolvemos*. A veces no es tan fácil decidir qué debe contener la especificación de un algoritmo. En principio, ante un problema que hay que resolver, podemos seguir las indicaciones que exponemos a continuación:

- En la precondición describiremos todo lo que tenga que ver con las condiciones en las que nuestro algoritmo debe poder resolver el problema.

- En la postcondición pondremos aquello que queremos obtener, el resultado que hay que obtener de nuestro algoritmo para que quede resuelto el problema. Prácticamente siempre tendremos que expresar la postcondición en términos de lo que hay en la precondición; es decir, tendremos que relacionar el estado final con el estado inicial.

Siguiendo con el ejemplo de la lavadora, si ponemos la lavadora en marcha y la ropa está amontonada al lado de la lavadora, el algoritmo se ejecutará igualmente pero no resolverá el problema, pues cuando la lavadora haya acabado, la ropa seguirá amontonada e igual de sucia al lado de la lavadora (además, la lavadora quizá se habrá estropeado, pero éste es otro tema).

Así pues, hay ciertas condiciones que el estado inicial debe cumplir porque el algoritmo que utiliza la lavadora para lavar ropa resuelva el problema. Podríamos tener, por ejemplo, la siguiente precondición:

> { Pre: la ropa está dentro de la lavadora y la lavadora está conectada al suministro de luz y de agua. Además, el cajoncito destinado al jabón está lleno de jabón. }

Si además queremos que la ropa salga suave, se tendría que añadir que "el cajoncito destinado al suavizante está lleno de suavizante".

Una vez se ha cumplido todo esto, podemos poner en marcha la lavadora. Ésta ejecutará el algoritmo que dará como resultado que la ropa quede limpia (¡después de un cierto tiempo, por supuesto!). Si alguna de las condiciones de la precondición no se cumplen, podemos poner en marcha la lavadora (por lo menos, intentarlo) pero no tenemos ninguna garantía de que la ropa se lave.

En la descripción del estado final, tendremos que decir que la ropa que inicialmente teníamos sucia en el tambor, ahora está limpia. Fijaos en que no es suficiente con decir que la ropa que hay dentro de la lavadora está limpia. Tenemos que decir que *la ropa es la misma que había en el estado inicial*. Si no, observad el siguiente algoritmo:

> "Abrir la puerta de la lavadora. Tomar la ropa sucia y tirarla por la ventana. Ir hasta el armario, coger ropa limpia. Si no queda, ir a la tienda y comprarla (podríamos comprar para varias veces y dejar una parte en el armario). Poner esta ropa dentro de la lavadora. Cerrar la puerta de la lavadora."

Este algoritmo resuelve el problema en que pasamos de tener ropa sucia dentro del la lavadora a tener ropa limpia, pero no es esto lo que queremos. Lo que queremos es lavar la ropa sucia que teníamos inicialmente.

Siempre tenemos que prestar atención al hecho de relacionar la descripción que damos del estado final con la descripción del estado inicial.

Podríamos dar una postcondición como ésta:

> { Post: la ropa que inicialmente estaba dentro de la lavadora, continúa dentro, pero ahora está limpia. La ropa no está desgarrada. }

Si además especificásemos el algoritmo de una lavadora secadora (y, por tanto, estaríamos hablando de otro problema que hay que resolver), tendríamos que añadir: "y la ropa está seca".

El comentario "la ropa no está desgarrada" es importante, ya que si no apareciese, una lavadora que desgarrara la ropa (además de lavarla) sería una lava-

Fijaos en que...

... si consideramos los algoritmos como funciones matemáticas (si pensáis un poco, veréis que, de hecho, lo son o se pueden ver como tales), la precondición nos determina el dominio de aplicación de la función y la postcondición nos dice cuál es el resultado de la función.

dora válida para la especificación (pero, evidentemente, no lo sería para lavar ropa en la vida real, ¡que es lo que estamos intentando especificar!).

2.3. Elementos de la especificación

La especificación de un algoritmo consta de cuatro partes:

1) La declaración de variables
2) La precondición
3) El nombre del algoritmo
4) La postcondición

> La *declaración de variables* define el conjunto de variables que forman parte del entorno del algoritmo y nos indica de qué tipo son; la *pre* y *postcondición* son la descripción del estado inicial y final de este entorno; y el *nombre del algoritmo* se corresponde con el nombre que le damos al algoritmo que estamos especificando, y es aconsejable que sea lo más significativo posible.

De esta forma, lo que estamos diciendo con la especificación es lo siguiente:

1) Dadas unas variables con las que tenemos que ser capaces de representar un problema y su solución,

2) y dadas las condiciones del estado inicial (indicadas en la precondición)

3) tenemos que diseñar un algoritmo tal que,

4) sea capaz de obtener el estado final deseado, descrito en la postcondición.

Ahora ya tenemos todos los ingredientes necesarios para poder especificar algoritmos. Así pues, veamos un ejemplo: la especificación de un algoritmo que calcula el factorial de un número entero.

Recordad que...

... el factorial está definido únicamente para números naturales y que el factorial de un número es el producto de todos los números naturales desde 1 hasta este número. El factorial de 0 está definido como 1.

En primer lugar, debemos decidir cómo representamos el problema y su solución. En este caso, utilizaremos una variable entera para representar el número cuyo factorial queremos calcular y otra (también entera) para guardar el factorial de este número. Las llamaremos respectivamente *n* y *fact*.

En la precondición tendremos que explicar cuáles son las condiciones en que el algoritmo se podrá ejecutar y nos dará un resultado correcto. En este caso, deberíamos decir que el número a partir del cual calculamos el factorial (es decir, *n*) es mayor o igual que 0. Lo que hacemos en la precondición es definir sobre qué dominio nuestro algoritmo debe ser capaz de resolver el problema.

En la postcondición deberíamos decir qué condiciones cumple el estado final; es decir, aquel en el que ya hemos solucionado el problema. En este caso, la condición que se debe cumplir es que en *fact* tenemos el factorial de *n*.

Veamos pues, en qué formato expresaremos a partir de ahora las especificaciones:

n, fact: **entero**;
{ Pre: *n* tiene como valor inicial *N* y $N \geq 0$ }
factorial
{ Post: *fact* es el factorial de *N* }

En la especificación anterior utilizamos la N mayúscula para designar el valor inicial de la variable *n*. Este "truco" es necesario, dado que el valor de la variable *n* puede cambiar mientras se ejecuta el algoritmo (nadie nos garantiza que el algoritmo factorial no modifique *n*). Por este motivo, para las variables para las que en la postcondición necesitamos conocer su valor en el estado inicial utilizaremos este "truco". Por convención, siempre utilizaremos como nombre del valor inicial de una variable el nombre de la misma variable en mayúsculas.

A partir de ahora abreviaremos frases del estilo de "*n* tiene como valor inicial *N*" por expresiones equivalentes como "$n = N$".

En la postcondición decimos simplemente que la variable *fact* tiene como valor el factorial de *N*. No es necesario decir que el factorial de un número es el producto de los números naturales desde 1 hasta este número, ya que se supone que nosotros ya lo sabemos (y por tanto, no es necesario que nos lo digan para que podamos resolver el problema). Y si nos lo dijeran no estaría mal; simplemente nos estarían dando más información de la necesaria.

Para cada especificación existe un conjunto de algoritmos que la cumplen o se adaptan. Es decir, que, dado un problema, normalmente lo podremos resolver de varias formas. Todas serán correctas y, por lo tanto, conseguirán, por lo menos, este objetivo de los cuatro marcados en el documento de introducción de la asignatura. Sin embargo, no todas conseguirán los otros tres objetivos: inteligible, eficiente, general. Por tanto, cuando diseñemos un algoritmo tendremos que ser capaces de diseñar un algoritmo correcto pero que cumpla a la vez los otros tres objetivos en la mayor medida posible.

2.4. Especificación y comentarios

Hasta ahora hemos hablado de la especificación como paso previo (y necesario) al diseño de un algoritmo. La especificación también nos puede servir, una vez hecho el algoritmo, para recordar claramente qué hace nuestro algoritmo sin la necesidad de repasar todo el código. Esto adquiere importancia a medida que aumenta la complejidad de los algoritmos que diseñamos. Deducir qué hace un algoritmo a partir del algoritmo mismo puede no ser rápido y nos podemos equivocar fácilmente.

Pensad que...

... cuando compramos una lavadora, podemos consultar la especificación de ésta en un catálogo y saber bajo qué condiciones funciona y cuáles son los resultados que se espera.

Aparte de todo esto, también es muy recomendable añadir comentarios dentro del algoritmo que nos permitan entenderlo fácilmente. Estos comentarios pueden conseguir uno de los siguientes efectos:

1) Especificar en qué situación o estado nos encontramos en un momento dado del algoritmo. Esto nos permitirá efectuar el seguimiento de los estados por los cuales va pasando un algoritmo de una manera cómoda.

2) Limitarse a ser comentarios puramente aclaratorios que hacen referencia a las construcciones algorítmicas que hemos utilizado para resolver un problema concreto, de manera que, más tarde, podamos entender rápidamente cómo actúa el algoritmo para resolver un problema. Estos comentarios no tienen nada que ver con la especificación, pero a menudo son muy útiles y es muy recomendable utilizarlos.

Este tipo de comentarios, igual que la pre y postcondición, se ponen entre llaves. De todas formas, por convención distinguiremos la pre y la postcondición añadiendo 'Pre:' y 'Post:' al principio de éstas, tal y como ya hemos hecho anteriormente. Tenéis que ser capaces de distinguir claramente entre la especificación y los comentarios puramente aclaratorios.

Los comentarios, aunque no formen parte del algoritmo propiamente, son bastante importantes para hacer más comprensible los algoritmos. De esta forma, pueden ayudar bastante en la fase de mantenimiento, en la que o bien nosotros mismos o bien otras personas deban entender un algoritmo desarrollado con anterioridad.

2.5. Ejemplos de especificación

En este apartado podéis encontrar ejemplos de especificación de problemas con una complejidad creciente.

2.5.1. Intercambio de dos variables

Queremos especificar un algoritmo que intercambie los valores de dos variables. Por este motivo, nuestro entorno estará constituido por estas dos variables, y tenemos que expresar, para ambas, que el valor final de una variable es el valor inicial de la otra. La especificación resulta como vemos a continuación:

x, y: **entero**
{ Pre: $x = X$ **e** $y = Y$ }
intercambiar
{ Post: $x = Y$ **e** $y = X$ }

2.5.2. Raíz cuadrada

x, raiz: **real**
{ Pre: $x = X$ **y** $X \geq 0$ }
raizCuadrada
{ Post: $raiz^2 = X$ **y** $raiz \geq 0$ }

Observad que en la precondición indicamos que no tiene ningún sentido calcular la raíz cuadrada de un número negativo. Por otra parte, un número positivo tiene una raíz negativa y una positiva. En la postcondición establecemos que el resultado corresponda a la positiva.

2.5.3. Raíz cuadrada entera

x, raiz: **entero**
{ Pre: $x = X$ **e** $X \geq 0$ }
raizCuadradaEntera
{ Post: $raiz^2 \leq X < (raiz + 1)^2$ }

2.5.4. Cociente y residuo

x, y, q, r: **entero**
{ Pre: $x = X$ **e** $y = Y$ y $X \geq 0$ **e** $Y > 0$ }
divisionEntera
{ Post: $X = q * Y + r$ **y** se cumple que $0 \leq r < Y$ }

2.5.5. Número de divisores

Queremos especificar un algoritmo que calcule el número de divisores positivos de un número positivo. En este caso, el entorno estará constituido por dos variables enteras: una donde tendremos el número y otra en cuyo estado final tendrá que constar el número de divisores de aquél.

x, nDivisores: **entero**
{ Pre: $x = X$ **e** $X > 0$ }
numeroDivisores
{ Post: *nDivisores* es el número de números enteros entre 1 **y** X que dividen X }

Notad que, puesto que nos piden el número de divisores sin especificar ninguna restricción, tenemos en cuenta también el 1 y X, los cuales son divisores de X seguros.

3. Estructuras algorítmicas

En este tema presentamos las construcciones básicas del lenguaje algorítmico que utilizaremos a lo largo de todo el curso para expresar los algoritmos.

3.1. Estructura general de un algoritmo

Veamos en primer lugar qué estructura tiene un algoritmo descrito en lenguaje algorítmico.

En todo algoritmo aparecerán las siguientes partes en el siguiente orden:

1) **Encabezamiento**. Nos sirve para identificar dónde empieza la descripción de nuestro algoritmo y para darle un nombre.

2) **Definición de constantes**. Inmediatamente después del encabezamiento definiremos las constantes que utilizaremos en nuestro algoritmo. En caso de que no utilicemos ninguna, esta parte no aparecerá.

3) **Definición de tipos**. En el primer apartado de este módulo habéis visto que las variables tienen un tipo determinado y que el mismo lenguaje algorítmico nos proporciona unos tipos básicos. A veces, sin embargo, nuestro algoritmo requerirá tipos más complejos que los que nos proporciona el lenguaje algorítmico. Si esto es así, tendremos que dar su definición justo después de las constantes. En caso de que no sea necesario definir nuevos tipos, no tendremos que poner nada en esta parte.

4) **Declaración de variables**. Aquí diremos qué variables utiliza nuestro algoritmo y de qué tipos son. La declaración de constantes y tipos se tiene que hacer antes porque, de este modo, aquí podremos utilizar los tipos y constantes que hayamos definido.

5) **Cuerpo del algoritmo.** Descripción en lenguaje algorítmico de las acciones que lleva a cabo el algoritmo.

6) **Final del algoritmo**. Nos sirve para tener claro dónde acaba el algoritmo.

A continuación tenéis un esqueleto de algoritmo donde aparecen sus diferentes partes.

```
algoritmo nombre;

    const
        nombreConstante1 : tipo = valorConstante1;
        nombreConstante2 : tipo = valorConstante2;
        ...
    fconst

    tipo
        nombreTipo1 = definicionTipo1;
        nombreTipo2 = definicionTipo2;
        ...
    ftipo

    var
        nombreVariable1 : tipoVariable1;
        nombreVariable2 : tipoVariable2;
        ...
    fvar

        accion1;
        accion2;
        ...

falgoritmo
```

Así, la palabra clave **algoritmo** se tiene que poner siempre al principio de un algoritmo; y junto con el nombre del algoritmo conforma el encabezamiento. La palabra clave **falgoritmo** se pone una vez ya hemos descrito todas las acciones del algoritmo.

Recordad que existe una serie de palabras clave que delimitan las partes del algoritmo y que nos ayudan a identificar fácilmente a qué corresponde cada elemento.

Las constantes siempre tienen que aparecer en medio de las palabras clave **const** y **fconst**. De esta forma, siempre que veamos estas palabras sabremos que todo lo que se encuentra en medio son definiciones de constantes.

Lo mismo sucede con las definiciones de tipos y las palabras clave **tipo** y **ftipo**, así como con las declaraciones de variables y las palabras clave **var** y **fvar**. Si alguna de estas partes no tiene ninguna definición, es decir, si no hay ninguna constante o ningún tipo o ninguna variable, tampoco es necesario que pongamos las palabras clave correspondientes.

Éstas no son todas las palabras clave que utilizaremos; a continuación encontraréis otras más que nos ayudarán, entre otras cosas, a definir las estructuras de control para combinar acciones sencillas en acciones más complejas.

3.2. Acciones elementales

El lenguaje algorítmico tiene únicamente una acción elemental: la acción de asignación. Más adelante, en el siguiente apartado, veremos cómo podemos definir nuestras propias acciones; estudiaremos una serie de acciones predefinidas (y no elementales) que nos ofrece el lenguaje para poder comunicar la ejecución del algoritmo con el exterior. De esta forma podremos obtener los datos necesarios para la ejecución del algoritmo o bien mostrar los datos correspondientes al resultado dado por el algoritmo.

3.2.1. La asignación

> La **asignación** es la forma que tenemos en lenguaje algorítmico para dar un valor a una variable.
>
> La expresamos de la siguiente forma:
>
> *nombreVariable* := *expresion*

Donde tenemos que *nombreVariable* es el identificador de una variable y *expresion* es una expresión. Se debe cumplir siempre que la variable y la expresión sean del mismo tipo. No podemos, por ejemplo, asignar valores booleanos a variables enteras o valores enteros a variables de tipo carácter.

La ejecución de la asignación consiste en evaluar la expresión *expresion*, lo que da como resultado un valor; posteriormente, se da a la variable *nombreVariable* este valor. Es decir, después de realizar la asignación, *nombreVariable* tiene como valor el resultado de evaluar *expresion.*

Dado que la asignación (y, en general, cualquier acción) modifica el entorno del algoritmo, podemos hablar de un "estado previo" a la ejecución de la acción, así como de un "estado posterior". Por tanto, podemos aplicar también los conceptos de pre y postcondición a una acción. De esta forma, podremos describir de una manera clara y sistemática el efecto que tiene la ejecución de una acción sobre el entorno.

Veámoslo para la asignación:



Para abreviar, utilizaremos ahora *x* como identificador de una variable y *E* como expresión.

{ el resultado de evaluar la expresión *E* es *V* }
x := *E*
{ *x* tiene como valor *V* }

Notad que...

... en la postcondición no aparece una asignación, sino la afirmación de una igualdad entre una variable y un valor.

A partir de ahora abreviaremos siempre la frase "*x* tiene como valor *V*" por un "*x* = *V*", expresión bastante más corta.

Ejemplos

En el caso de que tengamos una variable llamada *a*, que ésta sea de tipo carácter y que queramos asignarle el valor "*e*", podemos hacer:

{ }
a := 'e'
{ a = 'e' }

Si tenemos otra variable de tipo entero (pongamos por caso *d*), la expresión tendrá que ser de tipo entero.

{ }
$d := 3 * 4 + 2$
{ $d = 14$ }

Notad que...

... en estos dos casos, las expresiones están constituidas únicamente por constantes. Por tanto, independientemente del estado del entorno, tendrán siempre el mismo valor. Por eso, sea cual sea el estado previo del entorno, el resultado de la asignación será el mismo. Por este motivo, no hace falta establecer ninguna relación entre variables y valores en la precondición, que será siempre verdadera.

También podemos hacer asignaciones en las que intervengan variables en las expresiones correspondientes y, por tanto, dependan del estado del entorno.

Así, si *x* e *y* son dos variables reales, tenemos que:

{ $(y + 3)/2 = V$ }
$x := (y + 3.0)/2.0$
{ $x = V$ }

Donde V es el valor de la expresión, con la misma idea de los valores iniciales de las variables que poníamos en mayúsculas y que hemos introducido en el apartado anterior. Ésta es la aplicación directa de la especificación general de la asignación que hemos visto antes; pero estaréis de acuerdo con que también se puede expresar de la siguiente forma:

{ $y = Y$ }
$x := (y + 3.0)/2.0$
{ $x = (Y + 3)/2$ }

No podemos utilizar esta forma para la especificación genérica de la asignación, pues no sabemos qué variables intervienen en la expresión. Sin embargo, siempre que tengamos que especificar una asignación concreta lo haremos así, pues resulta más claro.

Por otro lado, el valor de la variable *y* no ha variado después de la asignación. Este hecho no queda reflejado en la postcondición, y es posible que nos interese (dependiendo del algoritmo concreto del que forme parte la asignación). Así pues, si nos interesase también este conocimiento, lo tendríamos que poner:

{ $y = Y$ }
$x := (y + 3.0)/2.0$
{ $y = Y$ y $x = (Y + 3)/2$ }

Es importante tener en cuenta que la expresión *E* se evalúa antes de dar el valor a la variable *x*. Así, si la variable *x* aparece también dentro de la expresión,

el valor que tendremos que utilizar para calcular la expresión será el que tenía en el estado previo a la asignación.

Veámoslo en el siguiente ejemplo. Supongamos que tenemos una variable entera llamada *n*:

$$\{ n = N \}$$
$$n := n * 2$$
$$\{ n = N * 2 \}$$

3.3. Composición de acciones

Ahora veremos cómo podemos combinar acciones elementales para construir algoritmos útiles, pues hasta ahora, los algoritmos que podemos construir consisten en una única acción elemental (asignación).

3.3.1. Composición secuencial

La primera y más obvia forma de composición que veremos es la composición secuencial de acciones. Consiste en ejecutar ordenadamente una secuencia de acciones: primero la primera, después la segunda, a continuación la tercera; y así hasta la última.

Lo expresamos de la siguiente forma:

$accion_1$;
$accion_2$;
..
$accion_n$

Escribimos una acción a continuación de la otra y separamos cada acción de la siguiente mediante un ";".

Hasta ahora habíamos hablado de acciones individuales. Éstas tenían un estado previo y un estado posterior (y por tanto, podíamos describirlas con lo que hemos llamado *precondición* y *postcondición* de la acción). Este estado posterior se conseguía después de la ejecución de una acción básica indivisible. Es decir, el entorno cambia en un único paso del estado inicial al estado final.

Una secuencia de acciones también tiene un estado previo y un estado posterior a su ejecución, estados que podemos describir mediante la pre y la postcondición de la secuencia de acciones; pero ahora, la transformación del estado inicial en el estado final no se produce en un solo paso:

- El estado previo a la ejecución de la secuencia de acciones se corresponde con el estado previo a la ejecución de la primera acción de la secuencia.

- Una vez ejecutada la primera acción obtenemos un nuevo estado, que es el estado posterior a la ejecución de la primera acción. Este estado es, a su vez, el estado previo a la ejecución de la segunda acción de la secuencia.

- Una vez ejecutada la segunda acción obtenemos un nuevo estado, que es el estado posterior a la ejecución de la segunda acción y, al mismo tiempo, el estado previo de la ejecución de la tercera acción.

- De esta forma, y haciendo lo mismo con el resto de acciones de la secuencia, llegaremos al estado previo de la última acción. Una vez ejecutada ésta, obtenemos el estado posterior a la ejecución de la última acción de la secuencia. Este estado se corresponde con el estado posterior a la ejecución de la secuencia de acciones.

Para llegar al estado posterior a la ejecución de la secuencia de acciones pasamos por una serie de estados intermedios. Dado el significado de cada una de las acciones que forman parte de la secuencia (determinada por su pre y postcondición), y dado también el estado previo a la ejecución de la secuencia, podremos determinar cada uno de estos estados intermedios, así como el estado posterior a la ejecución de la secuencia de acciones.

Puesto que no tenemos conocimiento de las acciones que formarán parte de la secuencia de acciones, no podemos decir gran cosa con respecto a su especificación, sólo lo que ya hemos dicho con respecto a los estados intermedios.

Lo representamos así:

$\{ P \}$
$accion_1;$
$\{ R_1 \}$
$accion_2;$
$\{ R_2 \}$
...
$\{R_{n-1}\}$
$accion_n$
$\{ Q \}$

Donde tenemos que P es la descripción del estado previo a la ejecución de la secuencia de acciones (precondición de la secuencia de acciones); R_i es la descripción del estado posterior a la ejecución de $accion_i$; y Q es la descripción del estado posterior a la ejecución de la sucesión de acciones (postcondición de la sucesión de acciones).

Cómo sean P, Q y R_i dependerá por completo de cuáles son las acciones que forman parte de la sucesión. Lo que es importante aquí es que cada una de es-

tas acciones nos determinará la relación entre R_{i-1} y R_i, y esto nos determinará la relación entre P y Q. !

Revisando el módulo "Introducción a la programación", podréis comprobar que el algoritmo de la lavadora consiste en una composición secuencial de acciones. Así pues, ahora ya podremos realizar los primeros algoritmos útiles.

Ejemplos

Intercambio de valores

Queremos dar una sucesión de acciones que intercambie los valores de dos variables enteras *x* e *y*. Supongamos que tenemos una tercera variable también entera (aux) que nos servirá para hacer el intercambio.

```
{ Pre: x = X e y = Y }
aux := x;
{ x = X e y = Y y aux = X }
x := y;
{ x = Y e y = Y y aux = X }
y := aux;
{ x = Y e y = X y aux = X }
{ Post: x = Y e y = X }
```

Notad que...

... a lo largo de todo este tema sólo veremos ejemplos de fragmentos de algoritmos, donde supondremos que tenemos definidas unas variables; y algunas de estas variables supondremos que ya han sido inicializadas. En el siguiente tema, donde se introduce el concepto de *entrada/salida*, ya veremos ejemplos de algoritmos completos.

Al ser la primera secuencia de acciones que vemos, y con vistas a que tengáis claro qué parte se corresponde con la especificación y qué se corresponde propiamente con el algoritmo, a continuación vemos la misma secuencia de acciones, pero ahora sin la parte correspondiente a la especificación/descripción de estados:

```
aux := x;
x := y;
y := aux;
```

Para resolver un problema,...

... nos pueden hacer falta más variables de las que nos hacen falta para especificarlo. Por este motivo, la descripción del estado final de un algoritmo puede contener más elementos que los indicados en la postcondición de la especificación. Lo que nos importa realmente es que todo lo que se dice en la postcondición de la especificación del problema sea cierto en el estado final.

Así pues, podríamos utilizar esta secuencia de acciones para dar un algoritmo (faltaría el encabezamiento, declaración de variables, etc.) que solucione el problema especificado en el apartado 2.5.1. Como podéis comprobar, la descripción del estado inicial de esta secuencia de acciones coincide con la precondición de aquel problema. Y la descripción del estado final, aunque no coincide (tiene más elementos, concretamente $aux = X$), sí satisface su postcondición. Notad que la variable *aux* es necesaria para resolver el problema, pero no toma parte en la especificación del mismo. Es lo que se conoce como *variable temporal*, o *variable auxiliar*.

Impuestos y salario neto

Dado el salario bruto de una persona, tenemos que proporcionar una secuencia de acciones que calcule el salario neto y los impuestos que debe pagar, sabiendo que esta persona paga un 20% de impuestos. Supongamos que disponemos de tres variables reales denominadas: *bruto*, *neto* e *impuestos*.

```
{ Pre: bruto = BRUTO }
neto := bruto * 0.8;
{ bruto = BRUTO y neto = BRUTO * 0.8 }
impuestos := bruto * 0.2;
{ Post: bruto = BRUTO y neto = BRUTO * 0.8 e impuestos = BRUTO * 0.2 }
```

Hemos resuelto este problema con una sucesión de dos acciones: primero calculamos el salario neto y después los impuestos. En la postcondición tenemos que el salario neto es el 80% del bruto y los impuestos, el 20%, tal como establece el enunciado.

Hemos incorporado al algoritmo la precondición y postcondición y la descripción de los estados intermedios (en este caso, sólo uno) para que veáis cómo las acciones de la secuencia determinan estos estados intermedios hasta que llegamos al estado final. Estas descripciones de estados, siempre entre llaves, no forman parte del algoritmo, sino que se incorporan a modo de comentario. !

Intereses

Dado un capital inicial invertido en un depósito financiero que da un 5% anual, dad una secuencia de acciones que calcule cuál es el capital que tendremos después de cuatro años (cada año, los intereses obtenidos pasan a aumentar el capital). Suponed que este capital inicial se encuentra en una variable real llamada *capital*; y que también queremos tener el capital final en esta variable.

```
{ Pre: capital = CAPITAL }
capital := capital * 105.0/100.0;
{ capital tiene el capital que tenemos después de un año, donde el capital inicial es CAPITAL }
capital := capital * 105.0/100.0;
{ capital tiene el capital que tenemos después de dos años, donde el capital inicial es CAPITAL }
capital := capital * 105.0/100.0;
{ capital tiene el capital que tenemos después de tres años, donde el capital inicial es CAPITAL }
capital := capital * 105.0/100.0;
{ Post: capital tiene el capital que tenemos después de cuatro años, donde el capital inicial es
CAPITAL }
```

Conseguimos nuestro propósito repitiendo cuatro veces la misma acción. Esta acción calcula el capital que tenemos al final de un año teniendo en cuenta el capital que teníamos a principios de año. Repitiéndola cuatro veces obtenemos el capital que tendríamos después de cuatro años.

3.3.2. Composición alternativa

Con lo que hemos visto hasta ahora, podemos construir acciones más complejas a partir de acciones más simples. Sin embargo, las acciones que ejecutamos siempre serán las mismas, independientemente de cuál sea el estado inicial. Con una restricción como ésta no podemos, por ejemplo, calcular el máximo de dos valores.

La composición alternativa nos permite decidir qué acciones ejecutaremos según cuál sea el estado. A continuación tenéis su sintaxis:

```
si expresion entonces
    accion_a
sino
    accion_b
fsi
```

Donde tenemos que *expresion* es una expresión de tipo booleano, por lo que a veces también se llama *condición*, y $accion_a$ y $accion_b$ son dos acciones (o sucesiones de acciones, da igual). Su ejecución consiste en evaluar la expresión; entonces, si ésta es verdadera, se ejecutará $accion_a$; y si es falsa, se ejecutará $accion_b$. Se llama *alternativa doble*.

Las palabras **si**, **entonces**, **sino** y **fsi** son palabras clave del lenguaje algorítmico que nos sirven para delimitar fácilmente cada una de las partes de la composición alternativa.

Existe una versión más reducida, donde no aparece la parte correspondiente al **sino**. En este caso, se evalúa la expresión, y si es verdadera, se ejecuta $accion_a$, y si es falsa, no se hace nada. Se llama *alternativa simple* y su sintaxis aparece a continuación:

```
si expresion entonces
    accion_a
fsi
```

Utilizaremos la composición alternativa cuando queramos ejecutar una acción u otra dependiendo de si se cumple una determinada condición o no; es decir, cuando queramos variar el flujo de ejecución de nuestro algoritmo.

Veamos cómo queda expresado el significado de la composición alternativa en su especificación:

```
{ P y el resultado de evaluar expresion es un cierto valor B (que puede ser
cierto o falso) }
si expresion entonces
 { P y B }
 accion_a
 { Q_a }
sino
 { P y no(B) }
 accion_b
 { Q_b }
fsi
{ Se cumple Q_a o bien se cumple Q_b }
```

Notad que...

... dado que *B* es un valor booleano { *P* **y** *B* } equivale a decir { se cumple *P* **y** *B* **es cierto** }; mientras que { *P* **y no**(*B*) } equivale a { se cumple *P* **y** *B* **es falso** }.

Al igual que sucedía en la composición secuencial, Q_a, Q_b y P dependerán de cuál sea el entorno del algoritmo y de qué elementos sean $accion_a$ y $accion_b$. Después de evaluar la expresión booleana, elegimos una rama u otra del **si**, pero el estado no se modifica. El estado sólo se ve modificado cuando pasamos a ejecutar $accion_a$ o bien $accion_b$.

En cuanto a la especificación de la versión sin **sino**, ésta se obtiene fácilmente a partir de la especificación que acabamos de ver eliminando la parte del **sino** y sustituyendo la postcondición final Q_b por P. Esto es así dado que la composición alternativa sin **sino** sería equivalente a una con **sino**, donde $accion_b$ fuese una acción que no hace nada.

Ejemplo

Máximo de dos números enteros

Queremos hacer el cuerpo de un algoritmo que calcule el máximo de dos números enteros. Supongamos que estos dos números enteros están en dos variables enteras x e y, y que queremos dejar el resultado en otra variable entera z.

Se trata de asignar a z el máximo de los dos valores (x e y). El problema es que, al hacer el algoritmo, no sabemos cuál será el valor más alto: x o y.

- Si supiésemos que x es el mayor, podríamos hacer directamente: $z := x$
- Si, en cambio, supiésemos que el mayor valor es y, podríamos hacer $z := y$
- Si ambos valores son iguales, cualquiera de las dos asignaciones nos sirve.

Entonces, dependiendo de si x es mayor que y, o bien al revés, tendremos que actuar de una forma u otra. Para poder dar un algoritmo que resuelva el problema tenemos que elegir desde dentro del algoritmo entre ejecutar una acción u otra. Esto lo podemos conseguir mediante la composición alternativa. Veamos cómo:

```
{ Pre: x = X e y = Y }
si x > y entonces
    { x = X e y = Y y x > y }
    z := x
    { x = X e y = Y y x > y y z = X. por tanto: z = maximo(X,Y) }
sino
    { x = X e y = Y y x ≤ y }
    z := y
    { x = X e y = Y y x ≤ y y z = y. por tanto: z = maximo(X,Y) }
fsi
{ Post: z = maximo(X,Y) }
```

Al igual que en los ejemplos del apartado anterior, hemos añadido al fragmento de algoritmo una descripción de todos los estados intermedios por los que puede pasar la ejecución del algoritmo. No es necesario que hagáis esto vosotros al diseñar los algoritmos. Lo que sí es necesario, como paso previo, es que especifiquéis el problema; es decir, que deis una precondición y postcondición globales que el algoritmo tendrá que cumplir.

El mismo algoritmo, ahora sin la descripción de los estados intermedios (pero con la pre y postcondición), nos quedaría:

```
{ Pre: x = X e y = Y }
si x > y entonces z := x
sino z := y
fsi
{ Post: z = maximo(X, Y) }
```

Fijaos en que lo importante en la composición alternativa es que tanto de una rama como de otra se pueda deducir la postcondición que queríamos conseguir; en este caso, que *z* tuviese como valor el máximo de *x* e *y*.

3.3.3. Composición iterativa

Con la composición alternativa ya podemos resolver un mayor número de problemas, pero todavía tenemos que ir más allá. Así, por ejemplo, imaginad que el lenguaje algorítmico o el ordenador no saben multiplicar dos enteros (en el supuesto de que no dispusiesen del operador ∗) y, en cambio, sí los saben sumar. Cuando tengamos que multiplicar dos enteros, le tendremos que explicar cómo lo tienen que hacer.

De este modo, para conseguir que un entero z sea igual a 10 ∗ *x*, podemos actuar de esta forma:

```
z := x + x + x + x + x + x + x + x + x + x
```

Y si la expresión que hay que asignar a z fuese 25 ∗ *x*:

```
z := x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x + x
```

Otra forma de hacerlo para el caso de z = 10 ∗ *x* sería:

```
z := 0;
z := z + x;
z := z + x;
z := z + x;
z := z + x;
z := z + x;
z := z + x;
z := z + x;
z := z + x;
z := z + x;
z := z + x;
```

Pesado, ¿verdad? Además, resolvemos casos particulares del producto, no un caso general como *x* ∗ *y*, donde tenemos que ser capaces de determinar cuántas veces se tiene que ejecutar la acción z := z + x desde el propio algoritmo, según el valor que tenga la variable *y* en aquel momento.

Podéis encontrar otro ejemplo en el ejercicio *Intereses* sobre la composición secuencial, en el que, a partir de un capital inicial, vamos acumulando intereses durante cuatro años. ¿Qué sucede si queremos calcularlo para 20 años? ¿Y para un número indeterminado de años que guardamos en otra variable?



Habéis visto este ejemplo en el apartado 3.3.1.

Para resolver este tipo de situaciones nos servirá la composición iterativa. La expresaremos de la siguiente forma:

mientras *expresion* **hacer**
    accion
fmientras

Donde tenemos que *expresion* es una expresión de tipo booleano y *accion* es una acción o sucesión de acciones. Las palabras **mientras**, **hacer** y **fmientras** son palabras clave del lenguaje algorítmico.

La ejecución de la construcción **mientras** consiste en ejecutar la acción mientras la evaluación de la expresión tenga como resultado **cierto**. En el momento en que la expresión pase a ser falsa, no hace nada más. Si inicialmente *expresion* es falsa, no se ejecuta *accion* ninguna vez.

Cada una de las ejecuciones de la acción se denomina **iteración**.

Especificación

Para especificar la composición iterativa se utiliza una propiedad o predicado especial llamado *invariante* (y que representaremos con *I*).

El **invariante** es una descripción de la evolución de los cálculos acumulados a lo largo de todas las iteraciones que hemos hecho hasta un momento determinado.

Lo importante de esta descripción es que nos tiene que servir para describir el estado en que nos encontramos cuando todavía no hemos hecho ninguna iteración, cuando hemos hecho una, cuando hemos hecho dos... y cuando las hemos hecho todas.

Ésta es la razón por la cual la propiedad invariante se denomina de esta forma: una misma descripción nos sirve para describir el estado antes y después de cada iteración. Así pues, el invariante se tiene que cumplir:

1) Justo antes de la construcción iterativa; es decir, en el estado previo a la composición iterativa (todavía no se ha hecho ninguna iteración).

2) En el estado previo a la ejecución de la acción en todas las iteraciones.

3) En el estado posterior a la ejecución de la acción en todas las iteraciones.

4) Una vez ya se ha acabado la ejecución de la composición iterativa; es decir, en el estado posterior de ésta (se han hecho ya todas las iteraciones).

No os preocupéis demasiado por el invariante, más tarde comentaremos algunos aspectos más sobre esto. Dado que el invariante se debe cumplir antes y después de cada iteración y que, además, la evaluación de la expresión booleana no puede modificar nunca el estado, la especificación de la composición iterativa nos queda:

{ I y el resultado de evaluar *expresion* es un cierto valor booleano B (que puede ser cierto o falso) }
mientras *expresion* **hacer**
    { *I* **y** *B* }
    accion
    { *I* }
fmientras
{ *I* **y** **no**(*B*) }

Una vez acabado el **mientras**, sabemos que no se cumple la condición *B*. Esto, juntamente con el invariante, nos tiene que permitir afirmar que hemos llegado al estado que deseábamos. Lo veremos a continuación con un ejemplo.

Ejemplos

Multiplicación de dos enteros

Volvamos al ejemplo motivador del inicio de este apartado. Ahora ya tenemos las herramientas para poder multiplicar dos enteros utilizando sólo sumas, y de una manera general. Aquí resolveremos el algoritmo para el caso en que la *y* es positiva. Fácilmente podemos ampliar el algoritmo utilizando el operador de cambio de signo para obtener un algoritmo que funcione para cualquier *y*. Como en los ejemplos anteriores, supondremos que ya tenemos los valores en las variables *x* e *y*, y que queremos guardar el resultado en una variable ya definida llamada *z*.

```
{ Pre: x = X e y = Y e Y ≥ 0 }
z := 0;
mientras y ≠ 0 hacer
   z := z + x;
   y := y – 1
fmientras
{ Post: z = X * Y }
```

Nos hemos ahorrado la descripción de los estados intermedios expresamente para poder formular la siguiente pregunta: ¿cuál es aquí la propiedad invariante?

Bien, sabemos que la propiedad invariante se tiene que cumplir al principio y al final de cada iteración. Podríamos elegir, por ejemplo $x = X$. Esta propiedad se cumple antes y después de cada iteración (de hecho, se cumple siempre, pues no modificamos su valor).

Sin embargo, ¿la propiedad $x = X$ nos describe los cálculos realizados? No, en absoluto. No decimos nada con respecto a la multiplicación que estamos realizando paso a paso. El invariante debe contener suficiente información para que nos sirva para deducir, a partir de I y **no**(B), la postcondición a la que queremos llegar (en este caso, $z = X * Y$). Evidentemente, con $x = X$ no lo conseguimos.

Una propiedad que sí nos describe los cálculos realizados es: $x = X$ **y** $z = x * (Y - y)$; y además se cumple antes y después de cada iteración. Podéis comprobar cómo ésta sí nos permite llegar a la postcondición esperada. Volvamos a ver la parte central del algoritmo anterior, pero ahora incorporando el invariante y el valor de la expresión B a la descripción de los estados intermedios:

```
{ Pre: x = X e y = Y e Y ≥ 0 }
z := 0;
{ x = X y z = x * (Y – y) e y ≥ 0 }
mientras y ≠ 0 hacer
  { x = X y z = x * (Y – y) e y > 0 }
  z := z + x;
  y := y – 1
  { x = X y z = x * (Y – y) e y ≥ 0 }
fmientras
{ Post: x = X y z = x * (Y – y) e y = 0; por tanto: z = X * Y }
```

Esta descripción del invariante sólo pretende que entendáis lo mejor posible la especificación de la composición iterativa, qué significa exactamente "propiedad invariante" y el porqué de ésta. Sin embargo, en ningún caso tendríais que encontrar vosotros mismos los invariantes de las composiciones iterativas que diseñéis (en asignaturas posteriores ya tendréis que batallar con ellos). De todos modos, a lo largo de los apuntes os encontraréis con invariantes muchas veces como complementos de la explicación de los algoritmos que vamos viendo.

El factorial

Veamos otro ejemplo en el que utilizaremos la composición iterativa. Queremos realizar un fragmento de algoritmo que, dado un número entero (que supondremos en la variable *n*), calcule el factorial, dejándolo en una variable llamada *fact*.

Hemos visto la especificación de este problema en el tema anterior. Para solucionar el problema, tenemos que multiplicar todos los números naturales entre 1 y el número cuyo factorial queremos calcular. Para hacer esto, tenemos que utilizar la composición iterativa. Supondremos que tenemos definida una variable extra para saber cuál es el número que estamos multiplicando en cada momento. Veámoslo:

```
{ Pre: n = N y N ≥ 0 }
fact := 1;
i := 1;
{ n = N y N ≥ 0 y fact es el factorial de i – 1 e i ≤ n + 1 }
mientras i ≤ n hacer
  { n = N y N ≥ 0 y fact es el factorial de i – 1 e i ≤ n }
  fact := fact * i;
  i := i + 1
  { n = N y N ≥ 0 y fact es el factorial de i – 1 }
fmientras
{ Post: n = N, N ≥ 0, fact es el factorial de i – 1 e i = n + 1, por tanto: fact es el factorial de N }
```

Notad que...

... la descripción del estado final obtenida a partir de { y **no**(B) } puede ser mucho más rica que la postcondición a la cual queremos llegar (la que nos indica **qué estamos haciendo**). Lo importante es que la postcondición **se pueda deducir** de la descripción del estado final.

En este ejemplo tenemos que volver a describir los estados intermedios mediante el invariante. Esto lo hemos hecho con una intención puramente didáctica, y ya no lo haremos tan detalladamente a partir de ahora. Sólo notad que, después de la composición iterativa, del hecho de que se cumpla el invariante y la condición del **mientras** sea falsa podemos deducir la postcondición a la que queremos llegar: que *fact* es el factorial de *N*.

Finalización de la composición iterativa

Un punto bastante importante cuando utilizamos la composición iterativa es saber con seguridad que se acabará en algún momento. Siempre que hacemos una iteración corremos el riesgo de hacerla mal y que no acabe nunca. Así, si en el ejemplo de la multiplicación de dos números nos olvidamos la acción $y := y - 1$, y valdrá siempre lo mismo y, por tanto, una vez entremos dentro del mientras, la acción $z := z + x$ se ejecutará indefinidamente.

Intuitivamente, tenéis que ser capaces de daros cuenta de que las modificaciones hechas por una iteración sobre el entorno nos llevan necesariamente a que nos queden menos iteraciones por hacer en la siguiente iteración.

Esto que vosotros haréis intuitivamente se puede demostrar formalmente mediante la ***función de cota***. La función de cota es una función matemática que tiene como dominio el conjunto de valores válidos para el conjunto de variables que intervienen en la iteración y como rango, los números enteros. Es decir, dado un estado, si le aplicamos la función de cota obtendremos un entero.

Esta función debe cumplir que el resultado de aplicarla antes de una iteración sea estrictamente mayor que el resultado de aplicarla después de la iteración. Además, si se cumple el invariante y B es verdadera, tiene que ser mayor que 0.

Si dada una composición iterativa somos capaces de encontrar una función de cota que cumpla estos requisitos, habremos demostrado que la construcción iterativa acaba.

Así, en el ejemplo del producto que hemos visto anteriormente, una función de cota válida (puede haber muchas) podría ser: $cota = y + 1$. En el caso del factorial, una función de cota correcta es: $cota = n - i + 1$.

En este curso no os pediremos nunca que encontréis el invariante ni la cota de una construcción iterativa, pero sí es importante que entendáis las explicaciones anteriores y que sepáis que tanto el comportamiento como la finalización de la ejecución de una construcción iterativa se pueden definir formalmente mediante los conceptos de *invariante* y *cota*.

Variantes de la composición iterativa

La construcción **mientras** nos permite expresar cualquier composición de acciones en la que tengamos que repetir una serie de cálculos. De todos modos, muchas veces esta repetición de acciones sigue un patrón concreto en el que utilizamos una variable que vamos incrementando (o disminuyendo) en cada

iteración hasta que llega a un valor dado. Para estos casos disponemos de una variante de la construcción iterativa que expresaremos así:

para *indice* := *valor inicial* **hasta** *valor final* [**paso** *incremento*] **hacer**
accion
fpara

Donde tenemos que la parte "**paso** *incremento*" es opcional y, si no aparece, se considera que incremento es igual a 1. Las palabras **para**, **hasta**, **paso**, **hacer** y **fpara** son palabras clave del lenguaje algorítmico.

Esta construcción se puede expresar con la construcción **mientras** que hemos visto anteriormente. Así que vosotros mismos podéis deducir el significado y la especificación:

```
indice := valor inicial;
mientras indice ≤ valor final hacer
  accion;
  indice := indice + incremento
fmientras
```

Notad, sin embargo, que...

... una construcción **mientras** no siempre podrá ser expresada mediante el uso del **para**. Podemos decir entonces que la construcción **mientras** es más general que la construcción **para**.

Fijaos que si el incremento fuese negativo, se debería cambiar la condición del bucle **mientras** a *índice* ≥ *valor final*.

Con esto, se podría reformular el algoritmo del factorial de la siguiente manera:

```
{ Pre: n = N y N ≥ 0 }
fact := 1;
para i := 1 hasta n hacer
  fact := fact * i;
fpara
{ Post: fact es el factorial de N }
```

Sólo nos hemos ahorrado dos líneas de código (la inicialización de la *i* y el incremento de ésta). Pero a veces nos resultará más cómodo utilizar esta construcción; especialmente para trabajar con tablas, elemento que será introducido en módulos posteriores. En general, siempre que se conozca el número de iteraciones, se utilizarà la construcción **para**.

4. Acciones y funciones

A veces nos encontraremos con problemas que serían más fáciles de expresar y resolver si dispusiéramos de acciones o funciones más convenientes que la simple asignación (la única acción que nos proporciona el lenguaje algorítmico) y de otras funciones que las de conversión de tipos.

De la misma forma que la notación algorítmica nos permite definir nuestros tipos (el tipo enumerativo, por ejemplo), podemos definir nuestras propias acciones y funciones.

Imaginad, por ejemplo, que nos dedicamos a resolver problemas del ámbito geométrico y que, a menudo, es necesario calcular el seno y el coseno de un ángulo. Si cada vez que necesitamos calcular el seno de un ángulo, tenemos que poner todas las acciones necesarias para poder realizar los cálculos con las variables correspondientes, los algoritmos serían largos, casi repetitivos, y con demasiados detalles que habría que tener en cuenta cuando los leyéramos. En cambio, si para expresar el algoritmo pudiéramos disponer de unas funciones que calculasen el seno y el coseno a partir de un ángulo, el algoritmo sería posiblemente más corto y, sobre todo, más claro, ya que nos concentraríamos más en el problema de geometría en sí, sin pensar en el problema del cálculo del seno o del coseno.

Puesto que la notación algorítmica carece de las funciones de seno y coseno pero nos permite definir funciones, podremos suponer que éstas existen, y más adelante diseñarlas preocupándonos únicamente una vez de cómo se calcula un seno y un coseno a partir de un ángulo dado. También podríamos necesitar acciones como girar puntos en el espacio, rectas en el espacio, etc., que harían más comprensible un algoritmo que resolviese un problema más general.

El hecho de que la notación algorítmica nos permita definir acciones y/o funciones nos será de gran ayuda, ya que nos permitirá enfrentarnos con problemas de mayor complejidad. Los algoritmos que solucionan estos problemas se formularán en términos de acciones más convenientes para la inteligibilidad de los algoritmos en sí y para su seguimiento. Más adelante desarrollaremos las acciones hasta llegar a la profundidad que requiere la notación algorítmica.

La posibilidad de crear nuevas acciones y funciones se puede considerar como un enriquecimiento o ampliación del repertorio de acciones disponibles del lenguaje algorítmico para enfrentarnos a problemas más complejos con más comodidad.

Aunque se puede pensar que las acciones y funciones nos servirán para ahorrarnos un buen número de instrucciones al diseñar un algoritmo, la importancia de las acciones y funciones radica en que son herramientas del lenguaje algorítmico que nos facilitarán la metodología que hay que seguir en el diseño de programas.

En el módulo "Introducción a la metología del diseño descendente" veremos cómo las acciones y funciones nos ayudan a resolver problemas complejos.

Por otra parte, al desarrollar los algoritmos podremos contar con algunas acciones y funciones que se consideran predefinidas, y que la mayoría de los lenguajes de programación proporcionan para facilitar la tarea de la programación, especialmente las acciones referentes a la entrada y salida de datos. También, a veces, podríamos desarrollar algoritmos a partir de una biblioteca de acciones y funciones ya desarrolladas por algún equipo de desarrollo de *software*. Así pues, las acciones y funciones permiten dividir el desarrollo de un algoritmo entre varias personas si se sigue una metodología y criterios adecuados para el fin que se persigue.

En algunas aplicaciones,...

... podemos disponer de una biblioteca de programas. En estos casos, las acciones y/o funciones han sido desarrolladas por otras personas para que nosotros podamos utilizarlas.

4.1. Acciones

Intuitivamente, una acción viene a ser una especie de subalgoritmo que puede ser utilizado desde cualquier punto de otro algoritmo o acción. De hecho, las acciones tienen la misma estructura que los algoritmos, pero un encabezamiento que se delimita por las palabras clave **accion ... faccion** en lugar de **algoritmo ... falgoritmo**.

Por tanto, dentro de la acción podemos definir además un entorno local o propio de la acción (secciones **const** ... **fconst**, **tipo** ... **ftipo**, **var** ... **fvar** definidas dentro de la acción). Se entiende por *entorno local* aquel conjunto de objetos que sólo puede ser utilizado por la acción que se desarrolla. Es decir, que otras partes del algoritmo no podrán utilizar este entorno.

Por ejemplo, imaginad que tenemos el siguiente enunciado:

Se desea hacer el intercambio entre los valores de las variables enteras *x* e *y*, *z* y *w* y *v* y *u*, respectivamente.

Es decir, nos pide que (especificando el problema...):

x, y, z, w, v, u: **entero**
{Pre: (*x* = *X* **e** *y* = *Y*) **y** (*z* = *Z* **y** *w* = *W*) **y** (*v* = *V* **y** *u* = *U*)}
tresIntercambios
{Post: (*x* = *Y* **e** *y* = *X*) **y** (*z* = *W* **y** *w* = *Z*) **y** (*v* = *U* **y** *u* = *V*)}

El problema del intercambio entre dos variables ha sido estudiado antes y, por tanto, podríamos escribir la siguiente solución:

{Pre: (*x* = *X* **e** *y* = *Y*) **y** (*z* = *Z* **y** *w* = *W*) **y** (*v* = *V* **y** *u* = *U*)}
aux := x;
x := y;
y := aux;
{(*x* = *Y* **e** *y* = *X*) **y** (*z* = *Z* **y** *w* = *W*) **y** (*v* = *V* **y** *u* = *U*)}
aux := z;
z := w;
w := aux;
{(*x* = *Y* **e** *y* = *X*) **y** (*z* = *W* **y** *w* = *Z*) **y** (*v* = *V* **y** *u* = *U*)}
aux := v;
v := u;
u := aux
{Post: (*x* = *Y* **e** *y* = *X*) **y** (*z* = *W* **y** *w* = *Z*) **y** (*v* = *U* **y** *u* = *V*)}

Si observamos el algoritmo resultante, nos damos cuenta de que contiene tres subalgoritmos parecidos que se diferencian sólo por algunas de las variables utilizadas. De hecho, estamos haciendo tres intercambios de dos variables dadas. Estamos repitiendo el mismo proceso de intercambio, pero primero aplicado a las variables *x* e *y*, después a las variables *z* y *w* y, finalmente, a las variables *u* y *v*.

Hubiese sido más fácil expresar el algoritmo de la siguiente forma:

{Pre: ($x = X$ **e** $y = Y$) **y** ($z = Z$ **y** $w = W$) **y** ($v = V$ **y** $u = U$)}
intercambia x e *y*
{($x = Y$ **e** $y = X$) **y** ($z = Z$ **y** $w = W$) **y** ($v = V$ **y** $u = U$)}
intercambia z y *w*
{($x = Y$ **e** $y = X$) **y** ($z = W$ **y** $w = Z$) **y** ($v = V$ **y** $u = U$)}
intercambia u y *v*
{Post: ($x = Y$ **e** $y = X$) **y** ($z = W$ **y** $w = Z$) **y** ($v = U$ **y** $u = V$)}

Y después hay que definir en qué consiste *"intercambia ... y ..."*.

Suponiendo que pudiésemos acceder a las variables del algoritmo que lo invocan desde la acción, podríamos diseñar esta acción como:

```
accion intercambia
    var
        aux: entero
    fvar
        aux := x;
        x := y;
        y := aux;
faccion
```

Muchos lenguajes de programación...

... permiten utilizar dentro de las acciones, variables que están declaradas en el algoritmo principal o en otros lugares del programa (variables globales). Como se puede deducir, es una mala costumbre utilizar esta posibilidad, ya que se producen programas poco inteligibles y muy difíciles de seguir.

Si suponemos además que las variables *x* e *y* están declaradas en el algoritmo que invoca la acción, tenemos otro problema: ¿cómo intercambiaremos las variables *z* y *w*, *v* y *u?*, ¿poniéndolas en *x* e *y* antes de la llamada? ¡Demasiado complicado!

Además, para diseñar esta acción tenemos que saber qué declaración de variables *x* e *y* ha hecho el algoritmo que la utiliza.

Por tanto, se da una dependencia demasiado grande entre el algoritmo o acción que la llama (*x* e *y*) y la acción intercambia. Tendríamos que diseñar siempre conjuntamente la acción y cualquier algoritmo que la quiera utilizar, y lo que queremos es poder definir una acción de intercambio más general que se pudiese aplicar a dos variables cualesquiera y que no dependiese del entorno particular desde donde se invoca.

En el ejemplo que estamos desarrollando, *tresIntercambios*, el algoritmo informal en que hemos puesto acciones del tipo "intercambia ... y ..." hemos expresado qué queríamos hacer y a qué variables afectaba lo que se quiere hacer. La primera vez queremos intercambiar *x* e *y*; en la segunda, el intercambio se hace sobre *z* y *w*, etc. Por tanto, queremos diseñar una acción general de intercambio de dos valores que se pueda aplicar de manera directa a cualquier entorno que la necesite.

Estos problemas expuestos se solucionan mediante el uso de parámetros.

4.2. Parámetros

Lo que necesitamos al construir y definir una acción es la posibilidad de expresar, dentro del cuerpo de la acción, acciones que actúen sobre objetos genéricos y, posteriormente, asociar estos objetos genéricos a objetos concretos definidos en el entorno desde el cual se utilice la acción.

Pensad en cómo trabajamos...

... las funciones matemáticas. Por ejemplo, si queremos conocer valores de la función $f(x) = x * x + 4$, tenemos que sustituir el símbolo x (el argumento o parámetro de la función) por un valor concreto en la expresión y, posteriormente, evaluarla. Así, $f(4) = 20$. De forma análoga, parte del cuerpo de la acción está expresado en términos de unos parámetros que tendrán que ser sustituidos por objetos reales.

En el ejemplo anterior nos iría bien poder indicar que *x* e *y* no son objetos reales, sino que sólo los utilizamos para formular cómo se hace el intercambio de dos variables, dejando pendiente para más adelante quién será realmente *x* y quién será realmente *y*.

Estos objetos genéricos (*x*, *y* en el ejemplo) se denominarán **parámetros formales de la acción**. Entonces decimos que la acción está parametrizada, es decir, que para poder realizarla tenemos que asociar antes estos parámetros a objetos concretos.

Por otra parte, los parámetros formales nos irán bien para expresar lo que queríamos de forma genérica, que después será aplicable al entorno que convenga en cada caso concreto.

Indicaremos en el encabezamiento de la acción cuáles son los parámetros formales con los que trabaja. Por tanto, la sintaxis del encabezamiento de una acción será:

accion *nombre* ($param_1$, $param_2$, ... , $param_n$)
  ... *cuerpo de la acción*
faccion

Donde *nombre* es el nombre que identifica la acción, y $param_i$ son parámetros de la acción.

Siguiendo nuestro ejemplo, la acción que necesitamos tiene el siguiente aspecto:

En este ejemplo...

... vemos el uso de **entsal** en los parámetros de la acción. Veremos el significado de esta palabra cuando estudiemos los tipos de parámetros que existen.

```
accion intercambia(entsal x: entero, entsal y: entero)
    var
        aux: entero
    fvar
        aux := x;
        x := y;
        y := aux;
faccion
```

Los parámetros formales de la acción son *x* e *y*. A pesar de que se denominen *x* e *y*, no tienen nada que ver con las variables *x* e *y* del algoritmo. Se trata de una coincidencia de nombres que no guardan relación entre sí.

Una vez definida la acción sobre unos parámetros formales, sólo es necesario indicar en la acción, en el momento de llamarla desde el algoritmo, sobre qué objetos reales debe actuar.

Por tanto, para utilizar una acción ya definida dentro de un algoritmo o dentro de otra acción, tendremos que escribir su nombre seguido de los objetos del entorno sobre los que queremos que actúe la acción.

Entonces diremos que **invocamos** o **llamamos** una acción. Para invocar la acción desde dentro de un algoritmo o de otra acción, lo expresaremos así:

$$nombreAccion(obj_1, obj_2, \dots , obj_n)$$

Donde obj_i es una variable, constante, o una expresión del entorno definido en el algoritmo.

Por medio de los parámetros, la acción se puede comunicar con los objetos del entorno que lo ha invocado.

Siguiendo el ejemplo, en el algoritmo, la invocación formal de las acciones para indicar a la acción parametrizada *intercambia* qué objetos reales utilizará se hará como se sigue:

```
{Pre: (x = X e y = Y) y (z = Z y w = W) y (v = V y u = U)}
intercambia(x, y);
intercambia(z, w);
intercambia(v, u);
{Post: (x = Y e y = X) y (z = W y w = Z) y (v = U y u = V)}
```

En la primera invocación, *x* e *y* son los **parámetros actuales** de la acción *intercambia*, en contraste con los parámetros formales, ya que éstos son los objetos concretos sobre los que se aplicará la acción. En la segunda, *z* y *w* son los parámetros actuales. Por tanto, los parámetros formales son los parámetros que necesitamos para definir (formalizar) la acción. Los parámetros actuales o reales son los objetos que se utilizan en la invocación.

La invocación *intercambia*(*z*,*w*) es equivalente a:

```
aux := z;
z := w;
w := aux;
```

O, dicho de otra forma,

```
{z = Z y w = W}
intercambia(z,w)
{z = W y w = Z}
```

Como podéis observar, la correspondencia entre parámetros actuales y formales sigue el orden textual de definición. En nuestro ejemplo, *z* se corresponde con el parámetro formal *x*, y *w* con el parámetro formal *y*.

El tipo definido para cada parámetro formal tiene que coincidir en lo que tenga cada parámetro actual. Así pues, en el ejemplo de intercambio no se pueden poner como parámetros actuales variables de tipo carácter, ni cualquier otro

tipo que no sea el entero. Se pueden definir en cada acción parámetros de diferente tipo (nuestro ejemplo no lo hace).

Los parámetros pueden ser de diferentes tipos según qué uso se tenga que hacer de ellos desde la acción.

Los parámetros se pueden clasificar en:

- **Entrada.** Sólo nos interesa consultar su valor.
- **Salida.** Sólo interesa asignarles un valor.
- **Entrada/salida.** Nos interesa consultar y modificar su valor.

Para indicar que un parámetro es de entrada, utilizaremos la palabra **ent**, mientras que si es de salida, usaremos la palabra **sal**, y si es de entrada/salida, la palabra **entsal**.

- Si un parámetro es de entrada, sólo nos interesa su valor inicial. Posteriormente podemos modificar este valor dentro de la acción, pero esto no afectará en ningún caso a los parámetros actuales.

- Si un parámetro es de salida, su posible valor inicial no podrá ser leído por la acción. Esto quiere decir que la inicialización de este parámetro tiene que hacerse en la acción. Una vez ejecutada la acción, el valor del parámetro actual correspondiente se corresponderá con el valor final de sus parámetros formales.

- Si el parámetro es de entrada/salida, el valor del parámetro podrá ser leído, y toda modificación de su valor tendrá efecto con los parámetros actuales.

La sintaxis de los parámetros $param_i$ en la sintaxis del encabezamiento de una acción es la siguiente:

para un parámetro de entrada es:	**ent** *<nombre> :<tipo>*
para un parámetro de salida es:	**sal** *<nombre> :<tipo>*
y para un parámetro de entrada/salida es:	**entsal** *<nombre> :<tipo>*

Si un parámetro formal es de entrada, el parámetro actual correspondiente podrá ser una variable, una constante o una expresión. Si un parámetro formal es de salida o de entrada/salida, el parámetro actual correspondiente tendrá que ser una variable.

Por ejemplo, si tenemos definido el siguiente encabezamiento de acción:

accion miraQueHago(**ent** valorEntrada : **entero**, **sal** resultado: **caracter**)

Podemos invocarla de las siguientes formas:

```
var
    numero: entero;
    miCaracter: caracter;
fvar

...
miraQueHago(numero, miCaracter);
```

...

o bien:

```
var
    x, y, contador: entero;
    miCaracter: caracter;
fvar
...
miraQueHago(contador * (x – y) div 100, miCaracter);
```

...

o bien:

```
var
    x, y, contador: entero;
    miCaracter: caracter;
fvar
...
miraQueHago(50, miCaracter);
```

...

Lo que sería incorrecto es:

```
...
miraQueHago(50, 'A')
...
```

Ya que 'A' es una constante del tipo carácter. Si es una constante, por definición no puede variar su valor, y esto contradice el hecho de que el parámetro formal correspondiente es de salida (puede modificarse el valor del parámetro).

4.3. Funciones

También tenemos la posibilidad de definir funciones. Las funciones sólo pueden aparecer en las expresiones. Por tanto, no se pueden invocar por sí solas.

Además, los parámetros formales (argumentos) de la función que se define sólo pueden ser de entrada. El efecto de invocar una función dentro de una expresión es ejecutar un conjunto de acciones que calculen un valor; y finalmente, retornar este valor, que debe ser del tipo que se haya declarado en el encabezamiento.

Su sintaxis es:

funcion *nombre*($param_1$,$param_2$,...,$param_n$): *tipo*
...
...
retorna *expresion*;
ffuncion

Donde $param_i$ es *nombre*: *tipo*. O sea, puesto que los parámetros tienen que ser siempre de entrada, nos ahorramos especificar que lo son con la palabra reservada **ent**.

La expresión **:** *tipo* indicará el tipo de valor que devolverá la función.

Toda función tiene que acabar en una sentencia como "**retorna** *expresion*", en la que el valor resultante de la evaluación de *expresion* tiene que ser del tipo declarado en el encabezamiento.

Al igual que las acciones, puede tener un entorno local.

Notad, pues, que la diferencia entre las acciones y las funciones reside, por un lado, en la forma de invocarlas, y por otro, en las restricciones de sus parámetros. La invocación de una función siempre tiene que formar parte de una expresión, mientras que la de una acción no forma parte nunca de ésta. En una función, sus parámetros siempre serán de entrada, y el valor que retorna es el único efecto de la función. En una acción, los parámetros no tienen estas restricciones. !

Estas restricciones...

... sobre los parámetros de las funciones no son mantenidas por algunos lenguajes de programación. Utilizar en estos lenguajes de programación, funciones con parámetros de salida o entrada/salida hace que los programas escritos sean poco inteligibles, ya que nadie espera que se cambien los argumentos de una función dentro de una expresión.

Por ejemplo, podríamos definir como función el producto basado en sumas que se ha tratado en un apartado anterior como:

```
{ Pre: x = X e y = Y y x > 0 e y > 0 }
funcion producto (x: entero, y: entero): entero
    var
        z: entero;
    fvar
    z := 0;
    mientras y ≠ 0 hacer
        z := z + x;
        y := y – 1
    fmientras
retorna z
ffuncio
{ Post: producto(x, y) = X * Y }
```

Y si en algún punto de un algoritmo tuviésemos que hacer el producto de *a* y *b* y poner el resultado en *c*:

```
var
    a, b, c: entero;
fvar
```

```
...
    c := producto(a, b);
...
```

Y si tuviésemos que hacer el producto de $a * b * c * d$ y poner el resultado en *f*, llamaríamos a la función de la siguiente manera:

```
var
    a, b, c, d, f: entero;
fvar

...
f := producto(producto(producto(a, b), c), d);
...
```

También podríamos utilizar la función dentro de una expresión como:

```
var
    a, b, c: entero;
fvar

...
c := 4 + 3 * producto(a, b);
...
```

4.4. Acciones y funciones predefinidas

En la notación algorítmica tendremos algunas acciones y/o funciones ya predefinidas.

4.4.1. Funciones de conversión de tipos

Ya conocemos algunas funciones predefinidas del lenguaje algorítmico, son las funciones de conversión de tipos que hemos visto cuando hablábamos de los objetos en el primer apartado. Aquí tenéis las declaraciones de los encabezamientos de estas funciones:

- **funcion** *realAEntero*(*x:* **real**): **entero**
- **funcion** *enteroAReal*(*x:* **entero**): **real**
- **funcion** *caracterACodigo*(*x:* **caracter**): **entero**
- **funcion** *codigoACaracter*(*x:* **entero**): **caracter**

4.4.2. Acciones y funciones de entrada y salida de datos

Ahora que ya habéis visto cómo se diseña un algoritmo, puede ser que os preguntéis cómo un algoritmo puede comunicarse con su entorno. El lenguaje algorítmico dispone también de un conjunto de acciones y funciones predefinidas que permiten a los algoritmos recibir datos desde el dispositivo de entrada y enviar datos al dispositivo de salida. De esta forma, los algoritmos pueden recibir los datos con que tiene que trabajar y retornar los resultados obtenidos.

Estas funciones y acciones son las siguientes:

- Función que retorna un entero que ha sido introducido por el teclado del ordenador:

funcion leerEntero(): **entero**

- Función que retorna un real que ha sido introducido por el teclado del ordenador:

funcion leerReal(): **real**

- Función que devuelve un carácter que ha sido introducido por el teclado del ordenador:

funcion leerCaracter(): **caracter**

- Acción que visualiza por pantalla el valor del entero *e*:

accion escribirEntero(**ent** e: **entero**)

- Acción que visualiza por pantalla el valor del real *r*:

accion escribirReal(**ent** *r*: **real**)

- Acción que visualiza por pantalla el valor del carácter *c*:

accion escribirCaracter(**ent** *c*: **caracter**)

Por tanto, consultad ahora un algoritmo completo que ya incorpora la entrada y salida de datos sobre el producto de naturales:

```
algoritmo productoNaturales
    var
        x, y, z: entero;
    fvar
    x := leerEntero();
    y := leerEntero();
    { Pre: x = X e y = Y y x > 0 e y > 0}
    z := 0;
    mientras y ≠ 0 hacer
        z := z + x;
        y := y −1
    fmientras
    { Post: z = X ∗ Y }
    escribirEntero(z);
falgoritmo
```

O bien, también se podría escribir como:

```
algoritmo productoNaturales
    var
        x, y, z: entero;
    fvar
    x := leerEntero();
    y := leerEntero();
    { Pre: x = X e y = Y y x > 0 e y > 0}
    z := producto(x, y);
    { Post: z = X ∗ Y }
```

```
    escribirEntero(z)
falgoritmo

funcion producto(x: entero, y: entero): entero
    var
        z: entero;
    fvar
    { Pre: z = Z }
    z := 0;
    mientras y ≠ 0 hacer
        z := z + x;
        y := y –1;
    fmientras
    { Post: z = X * Y }
    retorna z;
ffuncion
```

Resumen

En este módulo se han introducido los conceptos básicos (algoritmo, entorno, estado, procesador, etc.) que utilizaremos a lo largo del curso. Con la introducción de la notación algorítmica, ahora seremos capaces de expresar nuestros algoritmos con una notación formal y rigurosa para evitar cualquier ambigüedad.

Una idea central del módulo es ver el algoritmo como el agente que transforma un estado inicial del entorno del algoritmo en un estado deseado (y final) del entorno, pasando progresivamente por estados intermedios.

Ahora, ya podéis construir algoritmos para problemas de pequeña escala. En concreto, siguiendo las pautas que os hemos recomendado, será necesario:

1) Entender el enunciado: Especificaremos el algoritmo que hay que diseñar. De alguna forma, se trata de una reformulación del enunciado con términos más cercanos a la algorítmica, de modo que eliminemos las dudas que inicialmente podamos tener. Es importante no preocuparnos de **cómo** lo haremos, sino de **qué** tenemos que hacer.

{ Pre:... }	→	Comentamos con precisión (especificamos) cuál es el estado inicial de partida.
Nombre del algoritmo...	→	Ponemos el nombre del algoritmo que hay que diseñar.
{ Post:... }	→	Comentamos con precisión (especificamos) cuál es el estado final después de que el algoritmo se ha ejecutado.

2) Plantear el problema: todavía no hemos hablado mucho de cómo hacerlo desde un punto de vista metodológico. Por lo que sabemos hasta ahora, la construcción de un algoritmo pasará por descubrir el orden temporal en que se compondrán las acciones con las estructuras elegidas para conseguir los objetivos planteados. El seguimiento de los diferentes estados intermedios por los que pasará el algoritmo antes de llegar al estado final nos indicará el orden en que se compondrán las diferentes acciones. Los comentarios que hacemos para especificar con precisión el estado del entorno en un instante dado, entre las acciones que hay que desarrollar, nos ayudará a razonar y construir la composición correcta de acciones.

3) Formular la solución: ya conocemos la sintaxis y el significado de cada uno de los elementos del lenguaje algorítmico que nos permitirá expresar con precisión y rigurosidad el algoritmo que resuelve el problema.

{ Pre: ... }	→	Comentamos con precisión (especificamos) cuál es el estado inicial de partida.
algoritmo *nombre*	→	Describimos el algoritmo mediante el lenguaje algorítmico.
const ... fconst	→	Declaramos los objetos constantes (**entero**, **real**, **carácter**, **booleano**).
tipo ... ftipo	→	Declaramos los tipos que necesitamos (enumerados).
var ... fvar	→	Declaramos las variables (**entero**, **real**, **carácter**, **booleano**).
...	→	Describimos la secuencia de acciones usando las tres estructuras que conocemos: secuencial (;), alternativa (**si ... fsi**) e iterativa (**mientras ... fmientras**, **para ... fpara**).
falgoritmo		
{ Post:... }	→	Comentamos con precisión (especificamos) cuál es el estado final después de que el algoritmo se ha ejecutado.

A veces, tendremos la necesidad de encapsular ciertas acciones dentro de una acción y/o función parametrizada. A tal efecto, disponemos de los siguientes elementos:

{ Pre: ... }	→	Comentamos con precisión bajo qué condiciones la acción cumplirá el efecto deseado.
accion *nombre*($param_1$, ..., $param_n$)	→	Definiremos el encabezamiento de la acción con sus parámetros formales e indicaremos su tipo con **ent**, **sal** o **entsal**.
<Declaración entorno>	→	Declaramos los objetos que hay que utilizar localmente mediante **const ... fconst**, **tipo ... ftipo**, **var ... fvar**.
...	→	Describimos la secuencia de acciones que hay que realizar usando ";", **si ... fsi**, **mientras ... fmientras**, **para ... fpara**.
faccion		
{Post:...}	→	Comentamos con precisión el efecto de la acción.

o

{ Pre: ... }	→	Comentamos con precisión bajo qué condiciones la función cumplirá el efecto deseado.
funcion *nombre*($param_1$,...,$param_n$) : *tipo*	→	Definiremos el encabezamiento de la función con sus parámetros formales de entrada.
<Declaración entorno>	→	Declaramos los objetos que hay que utilizar localmente mediante **const ... fconst**, **tipo ... ftipo**, **var ... fvar**.
...	→	Describimos la secuencia de acciones que hay que utilizar usando ";", **si ... fsi**, **mientras ... fmientras**, **para ... fpara**.
ffuncion		
{ Post:... }	→	Comentamos con precisión el efecto de la función.

4) Evaluar la solución: por el momento, tenemos que reflexionar a partir de la semántica de cada uno de los elementos del lenguaje algorítmico y, paso a paso, ver que su composición es la adecuada. Los comentarios insertados entre acciones nos ayudarán a reflexionar y razonar sobre la corrección del algoritmo.

Ejercicios de autoevaluación

1. Indicad cuáles de las siguientes expresiones son sintácticamente correctas y, en el caso de que sea así, en qué casos son semánticamente correctas.

a) $x + y + 5.345 - 2.0$
b) **no** ($p \leq 3$ **div** (4 + n))
c) 5 + Zape(10.0 ∗ (b **o** c))
d) ∗3/4 + 6
e) – – – – – – – – – – 6– – – – – –2

2. Dadas las declaraciones siguientes, indicad si son correctas sintáctica y semánticamente.

a)

var
    maderas, *ident*: **entero**;
    hecho, *canto*: **booleano**;
fvar

b)

const
    1mas8: **entero** = 9;
fconst

c)

var
    porciento, *cociente*, *residuo*, *respuesta*: **entero**;
    respuesta: **booleano**;
fvar

d)

var
    suma, *real*, *producto*: **entero**;
    encontrado: **booleano**;
fvar

e)

const
    IVA: 16;
fconst

f)

const
    pi: **entero** = 3;
fconst

g)

const
    base: **entero** = 2;
    grado: **real** = 10;
fconst
var
    cantidad, *numeroPiezas*: **entero**;
    litros, *peso*, *grado*, *volumen*, *area*: **real**;
fvar

h)

const
    final: caracter = f;
fconst

i)

```
const
  cierto: entero = 1;
fconst
```

j)

```
var
  fijo, variable: entero;
  interes: real;
  respuesta: caracter;
fvar
```

3. Sean *x*, *y*, y *z* variables enteras no negativas (que pueden ser cero o positivas) y *c* una variable de tipo carácter. Simplificad, cuando podáis, las siguientes expresiones. Indicad también si se puede producir error en alguna circunstancia.

a) *x* **mod** *y*, donde además se sabe que siempre se da $x < y$.
b) *x* **mod** (*y* **mod** *z*), y sabemos que se cumple *y* **mod** $z = 0$.
c) $x * x$ **div** $x * x + x$ **mod** $x = x * (x + 1 - x$ **div** $x)$.
d) (c = 'A') **y** (c = 'E') **y** (c = 'I') **y** (c = 'O') **y** (c = 'U').
e) $(x > y)$ **o** $(x = y)$ **o** $(x < y)$.

4. Formulad las sentencias siguientes en forma de expresión booleana para *a*, *b*, *c* y *d* enteros:

a) Los valores de *b* y *c* son los dos superiores al valor de *d*.
b) Los valores de *a*, *b* y *c* son idénticos.
c) Los intervalos cerrados [*a*, *b*] y [*c*, *d*] interseccionan.
d) Hay dos valores idénticos, y sólo dos, entre *a*, *b* y *c*.
e) Hay como máximo, dos valores idénticos entre *a*, *b* y *c*.
f) El valor de *b* está comprendido, estrictamente, entre los valores de *c* y *d*.

5. ¿Cómo se relaciona la postcondición de una especificación con su precondición?

6. ¿En qué se diferencia la descripción del estado final de un algoritmo de su postcondición?

7. Especificad el algoritmo de cambiar una rueda pinchada de un coche (de una manera informal pero manteniendo la estructura precondición-postcondición).

8. Especificad el algoritmo que sigue una persona para hacer zumo de naranja. Intentad que la especificación no dependa de las herramientas utilizadas. ¡Cuidado, recordad dejar limpio el espacio donde habéis hecho el zumo y las herramientas que tenéis que utilizar! (Hacedlo con la misma filosofía que el ejercicio anterior.)

9. Especificad un algoritmo que sume los cuadrados de los *n* primeros números naturales.

10. En el apartado donde se introduce la construcción iterativa **mientras**, se presenta como ejemplo un algoritmo que multiplica dos números: *x* e *y*. Repasad el algoritmo y contestad las preguntas siguientes:
a) ¿Qué pasaría en este fragmento de algoritmo si $y < 0$? (notad que este hecho contradice la precondición, que nos indica que el algoritmo únicamente nos dará un resultado correcto si la y es positiva).
b) ¿Y en el caso en que $y = 0$?
c) ¿Y si en lugar de la condición $y \neq 0$ ponemos $y > 0$?

11. Diseñad un algoritmo que calcule la potencia de un número entero positivo elevado a otro número entero mayor o igual que cero.

12. Dad un algoritmo que calcule la raíz entera de un número entero.

13. Proponed un algoritmo que, dado un número natural *n* que leeremos del teclado, escriba la suma de los cuadrados de *n* primeros números naturales (empezando por el 1).

14. Confeccionad un algoritmo que transforme una cantidad de segundos en días, horas, minutos y segundos.

15. Haced un algoritmo que, dado un momento de tiempo determinado por cuatro variables en las que incluyamos el número de días, horas, minutos y segundos transcurridos desde un momento inicial determinado, incremente en un segundo el tiempo transcurrido.

16. Dada la definición de la función *f* definida a continuación, indicad qué valores tendrán *x*, *a*, *c*, y *d* después de la llamada de la función *f*.

```
funcion f(a: entero, b: real, c: caracter;
        d: booleano) : booleano
    a := 3;
    b := 4.0;
    c := 'A';
retorna no (d o (a > 4) y (b < 3.5))
ffuncion
```

```
var
    d: entero; c: real; a, x: booleano
fvar
...
a := cierto;
c := 7.0;
d := 5;
x := f(d, c, '5', a);
...
```

17. Diseñad una acción que, dado un número de segundos, nos proporcione el equivalente en días, horas, minutos y segundos. Fijaos en que es parecido al enunciado del ejercicio 14, pero aquí se os pide una acción.

18. Rediseñad el ejercicio 14 teniendo en cuenta que disponéis de la acción anterior.

19. Diseñad una acción que, dados los coeficientes reales A, B y C de una ecuación de segundo grado, calcule sus raíces reales en el caso de que lo sean, o que indique que no lo son. Podéis suponer que tenéis la siguiente función:

{ Pre: $r = R$ }

funcion raíz(r: real): real

{ Post: $raíz(r) = +\sqrt{R}$ }

Recordad que las raíces de una ecuación $ax^2 + bx + c = 0$ se calculan mediante la siguiente fórmula:

$$\frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

20. Teniendo en cuenta la solución del ejercicio 19, diseñad un algoritmo que lea los tres coeficientes desde el teclado y muestre por la salida las soluciones, si se da el caso. En caso de que no haya solución, se mostrará 'N' en la salida.

21. Diseñad una función que, dado un número entero positivo, retorne el número de cifras significativas que tiene.

Solucionario

1.
a) La sintaxis es correcta. Si x e y son reales, será semánticamente correcta.

b) La sintaxis es correcta. Si p y n son enteros, será semánticamente correcta. Fijaos en que, por el orden de prioridades, la expresión es equivalente a **no** ($p \leq (3$ **div** $(4 + n))$).

c) Sintácticamente es correcta. Para que lo fuese semánticamente, *Zape* tendría que dar un resultado de tipo entero, pero la expresión que aparece como argumento de la función no es semánticamente correcta, ya que se intenta hacer el producto de un real con un booleano.

d) Es sintácticamente incorrecta, porque el operador $*$ es binario.

e) Es sintácticamente correcta, por la interpretación dual que se tiene del símbolo –, que puede ser cambio de signo o resta. Operando, tendremos $6 - (-2) = 8$.

2.
a) Es correcta.
b) Es incorrecta, ya que un nombre no puede comenzar por un dígito.
c) Es incorrecta, ya que la variable *respuesta* no puede ser a la vez entera y booleana.
d) Es incorrecta, ya que **real** es una palabra reservada de la notación algorítmica.
e) Es incorrecta, ya que no sigue la sintaxis de definición de constantes.
f) Es correcta.
g) Es incorrecta, ya que *grado* no puede ser a la vez constante y variable.
h) Es incorrecta, ya que no se respeta la sintaxis de valores de caracteres.
Tendría que decir: final: **caracter** = '*f*'
i) Es incorrecta, ya que **cierto** es palabra reservada de la notación algorítmica.
j) Es correcta.

3.
a) La solución es x. No hay ningún error, ya que y no puede ser cero si $x < y$.
b) Da error, ya que y **mod** $z = 0$.
c) Si la variable x contiene el valor 0, la expresión no se puede evaluar porque no se puede dividir por 0. Si nos garantizan que la variable x tiene un valor diferente de cero, podemos aplicar la simplificación siguiente: parentizamos primero según la prioridad de operadores, y después procedemos a simplificar.
$((((x * x) \textbf{ div } x) * x) + (x \textbf{ mod } x)) = (x * ((x + 1) - (x \textbf{ div } x)))$
$((x * x) + 0) = (x * ((x + 1) - 1))$
$(x * x) = (x * x)$
cierto

d) **Falso**, ya que c sólo puede tener un valor. Si, por ejemplo, fuese "A", no sería "E", y *falso* con cualquier elemento da **falso**.
e) **Cierto**, ya que siempre uno de los predicados será cierto, y **cierto** o cualquier cosa, es **cierto**.

4.
a) $(b > d)$ **y** $(c > d)$
b) $(a = b)$ **y** $(b = c)$
c) $(a \leq d)$ **y** $(c \leq d)$ **y** $(((a \leq c)$ **y** $(c \leq b))$ **o** $((c \leq a)$ **y** $(c \leq d)))$
o bien
$(a \leq b)$ **y** $(c \leq d)$ **y** $(c \leq b))$ **y** $(a \leq d)$
d) $((a = b)$ **y** $(b \neq c)$ **o** $(b = c)$ **y** $(a \neq b))$ **o** $((a = c)$ **y** $(b \neq c))$
e) **no** $((a = b)$ **y** $(b = c))$
f) $((c < b)$ **y** $(b < d))$ **o** $((d < b)$ **y** $(b < c))$

5. En la precondición aparecen los valores iniciales de las variables de entrada (por convención, estos valores iniciales los ponemos en mayúsculas). Entonces, en la postcondición expresamos las variables de salida en función de estos valores iniciales.

6. En la descripción del estado final intervienen todas las variables que se utilizan en el algoritmo para solucionar el problema. En cambio, en la postcondición aparecen únicamente aquellas variables que se corresponden con la solución del problema (ved, por ejemplo, el algoritmo del factorial en el apartado de la composición iterativa).

Evidentemente, lo que nos resultará más interesante es la postcondición.

7. Especificación del algoritmo para cambiar una rueda pinchada:

{ Pre: el coche está parado. El coche tiene una rueda pinchada. Disponemos del gato para cambiar la rueda y de la rueda de recambio (en buenas condiciones), así como de la llave para destornillar los tornillos de la rueda }

ruedaPinchada
{ Post: el coche tiene las cuatro ruedas infladas y en buenas condiciones }

Notad que no explicamos en absoluto el proceso seguido para cambiar la rueda, sino que sólo describimos el estado inicial y el estado final, con el problema solucionado.

8. Especificación del algoritmo para hacer zumo de naranja:

{ Pre: Disponemos de un mínimo de 3 o más naranjas y de un vaso }
hacerZumoNaranja
{ Post: El vaso contiene el zumo de las naranjas. Las pieles de la naranja están en la basura. Todos los utensilios y espacios utilizados estan limpios y ordenados }

9. Especificación del algoritmo de la suma de los *n* primeros cuadrados:
n, s: **entero**;
{ Pre: $n = N$ y $N > 0$}
SumaCuadrados
{ Post: *s* es la suma de los *N* primeros cuadrados $(1 + 4 + 9 + ... + N^2)$ }

10. Preguntas sobre el algoritmo que multiplica dos números:

a) La ejecución de la construcción **mientras** no acabaría nunca. La variable *y* empieza siendo negativa, y en cada iteración se va reduciendo; por tanto, no llega nunca a 0.

b) Si $y = 0$, el cuerpo del **mientras** no se ejecuta nunca. Entonces, el valor final de *z* queda como lo hemos inicializado; es a decir, 0. El resultado, es, por tanto, correcto.

c) Cambiando la condición del **mientras** por $y > 0$ conseguimos que la ejecución del **mientras** acabe siempre, incluso para valores iniciales de *y* negativos. En cambio, el resultado dado en estos casos en los que la *y* es negativa es siempre 0 (y no $x * y$ como querríamos) y, por tanto, es incorrecto.

11. Algoritmo que calcula la potencia:

```
algoritmo potencia

    var
        n, exp, pot: entero;
    fvar

    n := leerEntero();
    exp := leerEntero();
    { Pre: n = N y exp = EXP, y EXP ≥ 0; y o bien EXP ≠ 0 o bien N ≠ 0 }
    pot := 1;
    mientras exp > 0 hacer
        pot := pot * n;
        exp := exp – 1
    fmientras
    { Post: pot = N^EXP }
    escribirEntero(pot)
falgoritmo
```

12. Algoritmo que calcula la raíz entera:

```
algoritmo raizEntera

    var
        n, raiz: entero;
    fvar

    n := leerEntero();
    { Pre: n = N y N ≥ 0 }
    raiz := 0;
    mientras (raiz + 1) * (raiz + 1) ≤ n hacer
        raiz := raiz + 1;
    fmientras
    { Post: raiz * raiz ≤ N < (raiz + 1) * (raiz + 1); es decir, raiz es la raíz entera de N }
    escribirEntero(raiz)
falgoritmo
```

13. Algoritmo de la suma de cuadrados:

```
algoritmo sumaCuadrados

    var
        n, s, i: entero;
    fvar
```

```
    n := leerEntero();
    {Pre: n = N }
    i := 1; { Inicializamos i con el primer natural }
    s := 0; { Inicializamos la suma a 0 }
    mientras i ≤ n hacer
        s := s + i * i; { Sumamos el cuadrado i-ésimo }
        i := i + 1
    fmientras
    { Post: s es la suma de cuadrados entre 1 y N }
    escribirEntero(s)
falgoritmo
```

14. Conversión de segundos a días, horas, minutos y segundos:

```
algoritmo conversorHorario

    var
        dias, horas, minutos, segundos: entero;
    fvar

    segundos := leerEntero();
    { Pre: segundos = SEGUNDOS y SEGUNDOS ≥ 0 }
    minutos := segundos div 60; { obtenemos el total de minutos }
    segundos := segundos mod 60; { obtenemos los segundos }
    horas := minutos div 60; { obtenemos el total de horas }
    minutos := minutos mod 60; { obtenemos los minutos }
    dias := horas div 24; { obtenemos el total de los días }
    horas := horas mod 24; { obtenemos las horas }
    { Post: segundos < 60 y minutos < 60 y horas < 24 y segundos +
    (minutos + (horas + dias * 24) * 60) * 60 = SEGUNDOS }
    escribirEntero(dias);
    escribirCaracter('d');
    escribirEntero(horas);
    escribirCaracter('h');
    escribirEntero(minutos);
    escribirCaracter('m');
    escribirEntero(segundos);
    escribirCaracter('s');
falgoritmo
```

15. *Tick* horario:

```
algoritmo tickHorario

    var
        dias, horas, minutos, segundos: entero;
    fvar

    dias := leerEntero() ;
    horas := leerEntero() ;
    minutos := leerEntero() ;
    segundos := leerEntero() ;
    { Pre: dias = DIAS, horas = HORAS, minutos = MINUTOS y segundos = SEGUNDOS.
    Además se cumple que: DIAS ≥ 0 y 0 ≤ HORAS < 24 y 0 ≤ MINUTOS < 60 y 0 ≤ SEGUNDOS < 60 }
    segundos := segundos + 1;
    si segundos = 60 entonces
        segundos := 0;
        minutos := minutos + 1;
        si minutos = 60 entonces
            minutos := 0;
            horas := horas + 1;
            si horas = 24 entonces
                horas := 0;
                dias := dias + 1;
            fsi
        fsi
    fsi
    { Post: dias ≥ 0 y 0 ≤ horas < 24 y 0 ≤ minutos < 60 y 0 ≤ segundos < 60. Además,
    ((dias * 24 + horas) * 60 + minutos) * 60 + segundos = ((DIAS * 24 + HORAS) * 60 +
    + MINUTOS) * 60 + SEGUNDOS + 1 }
    escribirEntero(dias);
    escribirCaracter('d');
```

```
    escribirEntero(horas);
    escribirCaracter('h');
    escribirEntero(minutos);
    escribirCaracter('m');
    escribirEntero(segundos);
    escribirCaracter('s');
falgoritmo
```

16. La evaluación de la función es **no**(**cierto o falso y falso**), que da **no (cierto)**. Por tanto, como todos los parámetros de una función son de entrada,

x = **falso**
a = **cierto**
c = 7.0
d = 5

17. Haremos aquí una solución alternativa del problema teniendo en cuenta que un día tiene 86.400 segundos (24 ∗ 60 ∗ 60), una hora tiene 3.600 segundos (60 ∗ 60), y un minuto, 60 segundos.

```
{Pre: t = T y t ≥ 0}
accion segundosADiasHorasMinutosSegundos (ent t: entero,
    sal dias: entero, sal horas: entero,
    sal minutos: entero, sal segundos: entero)
dias := t div 86.400;
t := t mod 86.400
horas := t div 3.600
t := t mod 3600;
minutos := t div 60
segundos := t mod 60
faccion
{Post: T = 86 400 ∗ DIAS + 3.600 ∗ HORAS + 60 ∗ MINUTOS + SEGUNDOS y 0 ≤ SEGUNDOS y
SEGUNDOS < 60 y 0 ≤ MINUTOS y MINUTOS < 60 y 0 ≤ HORAS y HORAS < 24}
```

18.

```
algoritmo conversorHorario

    var
        dias, horas, minutos, segundos: entero
    fvar
    segundos := leerEntero();
    SegundosADiasHorasMinutosSegundos(segundos, dias, horas, minutos, segundos);
    escribirEntero(dias);
    escribirCaracter('d');
    escribirEntero(horas);
    escribirCaracter('h');
    escribirEnterominutos);
    escribirCaracter('m');
    escribirEntero(segundos);
    escribirCaracter('s');
falgoritmo
```

19.

```
{Pre: a = A y b = B y c = C}
accion RaizEcuacion(ent a : real, ent b : real, ent c : real,
                        sal tieneSolucion : booleano, sal x1 : real, sor x2 : real)
    var discr : real;
    fvar
    discr := b ∗ b – 4.0 ∗ a ∗ c;
    si discr ≥ 0 entonces
        x1 := (–b + raiz(discr))/(2.0 ∗ a);
        x2 := (–b – raiz(discr))/(2.0 ∗ a);
        tieneSolucion := cierto
    sino tieneSolucion := falso
    fsi
faccion
{Post: ((tieneSolucion = cierto) y (A ∗ x1^2 + B ∗ x1 + C = 0)
    y (A ∗ x2^2 + B ∗ x2 + C = 0) o (tieneSolucion = falso)}
```

20.

```
algoritmo RaizEcuacionSegundoGrado
   var
      a, b, c, x1, x2: real;
      tieneSol: booleano;
   fvar
   a := leerReal();
   b := leerReal();
   c := leerReal();
   {Pre: a = A y b = B y c = C y A ≠ 0}
   RaizEcuacion(a, b, c, tieneSol, x1, x2)
   {Post: (tieneSol = cierto) y (A * x1^2 + B * x1 + C = 0) y
     (A * x2^2 + B * x2 + C = 0) o (tieneSol : falso) }
   si tieneSol entonces
         escribirReal(x1);
         escribirReal(x2)
   sino
         escribirCaracter('N');
   fsi
falgoritmo
```

21.

```
{ Pre: n = N y N > 0 }
funcion nCifras(n: entero): entero
   var
      nDigitos, pot10: entero
   fvar;
   nDigitos := 1;
   pot10 := 10;
   mientras pot10 ≤ n hacer
      nDigitos := nDigitos + 1;
      pot10 := pot10 * 10
   fmientras
   retorna nDigitos
ffuncion
{ Post: sea x = nCifras(n), entonces podemos decir que (10^(x–1) ≤ N) y N < 10^x }
```

Glosario

asignación
Fija un valor a una variable dada. Es la acción más elemental del lenguaje algorítmico.

condición
Expresión booleana que, de acuerdo con el resultado de su evaluación (cierto o falso), controla la ejecución de una estructura algorítmica.

cuerpo de un algoritmo
Parte de un algoritmo correspondiente a las acciones.

cuerpo de una iteración
Parte de acciones que pueden repetirse.

encabezamiento
Identificación y caracterización general de un algoritmo, acción o función siguiendo la sintaxis del lenguaje algorítmico.

especificación
Reescritura del enunciado de un problema de forma que se relacionan las variables de entrada del cuerpo del algoritmo que hay que solucionar con las de salida. De esta forma se tiene claro cuáles son los objetivos del diseño que hay que efectuar y a partir de qué condiciones. Consta de precondición y postcondición.

estructura algorítmica
Forma que controla el orden en que se ejecutan las acciones. En la notación algorítmica hay tres: la secuencial, la alternativa, y la iterativa.

expresión
Combinación de operandos y operadores según unas reglas sintácticas.

invariante
Propiedad que se mantiene a lo largo de la ejecución de una estructura iterativa.

iteración
Estructura algorítmica que puede repetir un número de veces un conjunto de acciones.

palabra clave
Ved *palabra reservada*.

palabra reservada
Palabra que tiene un significado concreto para la notación algorítmica y que, por tanto, sólo se puede utilizar para su finalidad.

precondición
Afirmación sobre el estado del entorno en el instante anterior a la ejecución de un algoritmo o una acción.

postcondición
Afirmación sobre el estado del entorno en el instante en que se ha acabado de ejecutar una instrucción.

predicado
Expresión booleana.

Bibliografía

Castro, J.; Cucker, F.; Messeguer, X.; Rubio, A.; Solano, L.; Valles, B. (1992). *Curs de programació*. Madrid, etc.: McGraw-Hill.

Vancells, J. (1998). "Introducción a la algorítmica". En: Botella, P.; Bofill, M.; Burgués, X.; Franch, X.; Lagonigro, R. (1998). *Fundamentos de programación I*. Barcelona: Ediuoc.

Vancells, J.; López E. (1992). *Programació: introducció a l'algorísmica*. Vic: Eumo Editorial (Tecno-Ciència).

Nota

En la bibliografía referenciada no hay plena coincidencia por lo que respecta a la notación algorítmica propuesta. No obstante, los conceptos expuestos siguen, mayormente, lo expuesto en el módulo.